1 Pto

Computer Science Department
Computer Systems Laboratory
Information Systems Laboratory
Integrated Circuits Laboratory

STANFORD UNIVERSITY · STANFORD, CA 94305

85-0384

OK DTIC

TECHNICAL STATUS REPORT
1 May 1982 to 31 October 1982

Research in VLSI Systems
DARPA Contract No. MDA903-79-C-0680
DARPA Order No. 3773
Principal Investigators: J. Hennessy, T. Kailath
Sponsored by Defense Advanced-Research Projects Agency
Monitored by P. Losleben

Heuristic Programming Project and VLSI Theory Project
DARPA Contract No. MDA903-80-C-0107
DARPA Order No. 3423
Principal Investigators: E. A. Feigenbaum, B. G. Buchanan, J. D. Ullman
Sponsored by Defense Advanced-Research Projects Agency
Monitored by R. Ohlander

A Fast Turn Around Facility for Very Large Scale Integration (VLSI)
DARPA Contract No. MDA903-80-C-0432
DARPA Order No. 3709
Principal Investigator J. T. Meindl
Monitored by D. Adams

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the offical policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
MDA903-79-C-0680(2)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (*and Subtitle*)
Research in VLSI Systems
Heuristic Programming Project & VLSI Theory Project
A Fast Turn Around Facility for Very Large Scale Integration (VLSI)

5. TYPE OF REPORT & PERIOD COVERED
Technical Status Report
May 1982-October 1982

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(*s*)
J. Hennessy, R. Matthews, J. Newkirk,
J. Shott, J. Ullman, H. Brown

8. CONTRACT OR GRANT NUMBER(*s*)
MDA903-79-C-0680
MDA903-80-C-0107
MDA903-80-C-0432

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Computer Science Dept.; Computer Systems,
Information Systems, Integrated Circuits
Laboratories; Stanford University, Stanford, CA.

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
Arlington, Virginia 22209

12. REPORT DATE
November 1982

13. NUMBER OF PAGES
31

14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)
Office of Naval Research (Stanford Branch)
Stanford University
Stanford, California 94305

15. SECURITY CLASS. (*of this report*)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Unclassified; approved for general distribution.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (~~Continue on reverse side if~~ necessary and identify by block number)

Very Large Scale Integration,
Regular Expression Compilation,
MIPS: A VLSI Processor,
Relative Layout Tools
Graphics Architectures,
Computer Supported FTL,
Palladio: IC Designer's Assistant
Electron Bean Lithography
Micron CMOS,
Wafer FabricationFacility.

(JES) ⇐

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

Table of Contents

Executive Summary

The major progress of note for this period is as follows:

1. *Regular Expression Compilation.* The overall goal of the project is to develop silicon compilers that produce output comparable to hand designs. A compiler for translating a mixture of state machine definition and regular expressions into networks of PLA's or logic is working. In a number of tests the area required by the output was found to be no more than 50% over that of a hand design; in some cases the results are far closer than that.

2. *MIPS: A VLSI Processor.* MIPS (Microprocessor without Interlock between Pipe Stages) is a project to develop a high speed (> 1 MIP) single chip 32-bit microprocessor. The final test chips for the MIPS processor design were completed and will be submitted for fabrication on the November 82 MOSIS fabrication run.

3. *Relative Layout Tools.* Yale (Yet Another Layout Editor) is a symbolic layout editor that will run on the SUN and make the capabilities of SILT available in a graphics front-end. The first version of Yale was completed and documented.

4. *Graphics Architectures.* The IRIS is a high-resolution, high-performance, color-graphics workstation. It incorporates the Geometry Engine and utilizes the SUN processoir board. An IRIS prootype was designed and demonstration software was developed.

5. *Computer supported FTL launched.* We have completed the planning and exploratory stages of a project to provide extensive automation and computer support for the Fast Turnaround Laboratory. This ambitious interdisciplinary project (involving researchers from Computer Systems, Integrated Circuits, and Solid State laboratories) will provide control, documentation, training, portability, repeatability, and efficiency in the area of IC fabrication processes.

6. *Palladio: An IC Designer's Assistant* The Palladio system is a framework for experimentation with circuit design methodologies, knowledge-based expert system design aids and symbolic circuit simulation concepts. The major goal of the project is to develop an intelligent and integrated circuit design environment to assist in the full design, test and debug design cycle. During the past six months Palladio's basic system structure was completed and several prototypic system components were implemented.

7. *Electron Beam Lithography.* The Stanford MEBES machine has passed all on-site acceptance tests, including those relating to direct writing and registration accuracy. The MEBES machine is currently being used to fabricate masks for the Geometry Engine, several MIPS test chips, and the two-chip Medium Tester.

8. *2 Micron CMOS.* Test devices using Stanford's 2 μm CMOS process have been fabricated. This process features a 4 μm n-well and a 400 Angstrom oxide thickness. Two additional "tune-up" runs are in progress at the moment. The mask set for this run was written at Perkin-Elmer/ETEC before our machine passed acceptance, but plates were developed and etched at Stanford.

9. *Wafer Fabrication Facility.* The following pieces of equipment have been installed and characterized and are now being used in the fabrication of NMOS/CMOS wafers: Drytek 100 plasma etcher (Poly-Si and Si_3N_4); Tylan LPCVD (Poly-Si, Si_3N_4, and LT SiO_2); MTI Omni-Chuck resist processing; IPC Σ-80 plasma etcher (SiO_2); and Ultratech 900 1:1 projection aligner.

10. *Tri-Level Resist Technology.* A tri-level resist technology designed for direct-write E-beam lithography has been developed. The three layers are: 1.2 μm "super-hard-baked" AZ-1470 resist, 500 Angstrom sputtered or plasma-deposited poly-Si, and 0.4 μm PBS electron resist. The key step in this process is the use of O_2 reactive ion etching (RIE) to transfer to pattern from the thin PBS and poly-Si layers to the underlying "planarization" layer of AZ-1470 resist. Resist lines 0.5 μm wide separated by 0.5 μm have been produced by the highly anisotropic RIE.

11. *Testing for Process Control.* A number of test structures have been designed which allow statistical data to be gathered about many of the parameters affecting fabrication yield including step coverage, shorts (both level-to-level and on a single level), contact hole integrity, etc. In contrast to many of the "string" or "meander" structures, these test vehicles are addressable in such a way as to allow the position information of failures to be monitored.

12. *Electrically based layout system, Lava.* A rewritten version of Lava is again running test cases, including the 10,000-transistor serial memory. It seems stable enough to support further investigations, *e.g.*, composition of cells and logic-to-sticks conversion.

13. *Logic-to-sticks conversion, Dumbo.* Dumbo produced its first totally automatic layouts with reasonable area efficiency, using force-directed placement. Large cells still incur large area penalties, however.

14. *The MEDIUM tester chip set.* The MEDIUM tester chip set has been designed, laid out, and submitted for fabrication, along with some test chips. One of the two main chips has been partially tested, and it appears to be correct.

15. *Bulk CMOS.* We have implemented a CMOS design-rule checker based on the polygon package and distributed it to MIT and JPL. It was used to check pads, PLAs, and a counter submitted for fabrication; it has also checked layouts from MIT and Lincoln Labs.

Technical Progress

1 Design Description, Analysis, and Synthesis

1.1 Regular Expression Compilation

A system compiling regular expressions into PLA's or logic has been developed. The input language has been augmented recently to include state declarations when convenient; in the syntax, entering a state looks similar to the occurrence of an input symbol, while transfer to a state is akin to emitting an output symbol.

The regular expression language is translated to a nondeterministic finite automaton (NFA) language by one of two different compiler strategies, called "before" and "after." The former tends to minimize the number of rows of a PLA, while the latter tends to minimize the columns. Neither strategy dominates the other in tests, so both are made available as options for the user. → Key words →

The NFA's are compiled into networks of PLA's or Weinberger arrays (via S. Johnson's lgen language). The PLA's are optimized using GRY [Hemachandra 82]. Layout of PLA's is accomplished by PLAGEN, a routine written in CHISEL [Karplus 82]. The latter two facilities are the product of K. Karplus, a Hertz fellow whose DARPA support consisted of computer services.

Staff: L. Hemachandra, A. Karlin, H. Trickey, J. Ullman

Related Efforts: lgen (Bell Labs), SLIM (Stanford)

References: [Hemachandra 82, Floyd 82, Trickey 82, TrickeyUllman 82, Ullman 82a]

1.2 YALE

Yale (Yet Another Layout Editor) is a symbolic layout editor that will run on the SUN and make the capabilities of SILT available in a graphics front-end. YALE is presently being implemented on a combination of the SUN workstation and the VAX. It uses the SUN as an intelligent graphics workstation (no disc required); thus, this work is being carried out in collaboration with the Network Graphics project at Stanford. YALE is primarily a graphics interface to SILT, allowing the placement of *reference lines* graphically. It also allows textual or graphical specification of *constraints* and textual specification of expressions for computation of reference line placement.

Staff: J. Clark, T. Davis

Related Efforts: Daedalus and the Data Path Generator (MIT), Caesar (UCB).

References: [Davis, T. and Clark, J. 82]

1.3 SLIM

SLIM, Stanford language for Implementing Microcode, was initially implemented during an earlier contract and presented at the 1981 Caltech VLSI Conference. The goals of SLIM are to describe on-chip control as microcode, to simulate that microcode using a functional description of the chip components, and to generate a PLA implementation of the microcode. The initial SLIM implementation has been working since the end of 1980.

The current work on SLIM concentrates on the design of a state-assignment optimizer. A prototype optimizer, which saves an average of 15% of the minterms, has been developed. It needs further work to characterize its theortical properties and to make it more efficient on large PLA's.

Staff: J. Hennessy, L. Adams

Related Efforts: MacPitts (Lincoln Labs), SLANG (UCB) and SLAP (Brown University).

1.4 Palladio: An IC Designer's Assistant

The Palladio system [Brown 82] is a framework for experimentation with circuit design methodologies, knowledge-based expert system design aids and symbolic circuit simulation concepts. The major goal of the project is to develop an intelligent and integrated circuit design environment to assist in the full design, test and debug cycle. Palladio serves as the focus for the Knowledge-based VLSI Project (KBVLSI project), a collaborative activity between the Heuristic Programming Project, Stanford University, the VLSI System Design Area, Xerox Palo Alto Research Center and the Fairchild Laboratory for Artificial Intelligence Research.

1.4.1 Design Specification Editor

A circuit design process can be viewed as the creation of behavioral and structural specifications of the circuit. This usually involves a sequence of transformations from abstract specifications of the behavior and structure to more detailed implementation specifications. Palladio's design editor is an interactive graphics system for creating and editing circuit specifications at various levels of structural and behavioral detail. For example, in Palladio the structure of a circuit component can be simultaneously described in terms of gates, in terms of switches and in terms of a layout, and the component's behavior can be simultaneously described in terms of boolean logic or in terms of node value-strength pairs.

The level of specification to be used in performing a particular task is currently at the choice of the designer. We are working on design aids which will automatically select the most appropriate specification levels for the task at hand, for example, simulation.

During the past six months we have essentially completed the the underlying framework for Palladio's design editor and the graphical user interface to the editor. The graphical interface uses HILGA [Gerring 81], a high level graphics package for Interlisp-D.

Staff: H. Brown, G. Foyster, P. Gerring.

1.4.2 Design Specification Levels

One of the objectives of our project is to experiment with various circuit design specification levels. In Palladio a design specification level is primarily represented by structure and/or behavior specification languages (both graphical and textual) and by a set of composition rules for governing the recursive creation of composite components from the primitives of the specification language [Stefik 82a]. During the past six months we have implemented two specification levels.

The Clocked Primitive Switches (CPS) level is a structural specification level which includes both circuit and gate level specifications. In addition the graphical form of the CPS level can be used to describe a planar topology for the circuit.

Associated with the the CPS structural level are two behavioral specification levels. One is based on the usual boolean level of behavioral specification and the other is based on node value-strength pairs [Bryant 81]. The user interface form of both of these behavioral levels is production rules.

Staff: H. Brown, G. Foyster.

The Computational Blocks Abstraction (CBA) specification level is a level at which a designer can specify a digital system in terms of blocks containing data structures and operations.

Staff: C. Tong.

1.4.3 Design Simulator

During the past six months we have been working on a framework for multiple-level, rule-based simulator. The simulator is not tied to any particular technology or specification level. The primary idea is to exploit hierarchical design descriptions to help manage the simulation of complex systems. The simulator framework can be used to perform auto-validation of designs, goal-directed simulation, and symbolic simulation.

A preliminary version of the simulator framework has been implemented. This implementation was done in MACLISP using the Meta-level Representation System (MRS) [Genesereth 80]. MRS provides, in partiular, the simulator's inferencce mechanisms.

Staff: N. Singh (Stanford and Fairchild).

The simulator framework has been interfaced with the CPS specification level in Palladio, and a dynamic, graphical simulator display has been implemented.

Staff: G. Foyster.

1.4.4 Design Transformation

During the past year we have been working on design aids to assist in the transformation of an abstract design specification to a more detailed implementation. This work has involved research on a model for design centered around goals, alternative designs and tradeoffs [Tong 82]. This model views design as a dialectic between goals and design alternatives: goals are established, alternative designs are specified, and the goals are evaluated with respect to these alternatives and possibly revised in light of the alternatives. Knowledge of important tradeoffs among goals are used to guide the dialectic.

To support the complex and varying relationships among design entities a design knowledge representation language, FIRE, has been developed. FIRE is implemented in LOOPS [Bobrow 81].

Staff: C. Tong.

1.4.5 Programming Systems

Most of the Palladio system is implemented in LOOPS. LOOPS is a data and object oriented programming system integrated with Interlisp. In object oriented programming, behavior is determined by responses of instances of classes to messages sent between these objects with no direct access to the internal structure of an object. Data oriented programming is a dual of object oriented programming where behavior can occur as a side effect of direct access to object state.

During the past six months numerous enhancemens and extensions have been made to LOOPS. In particular, the capability for doing rule oriented programming has been added to LOOPS [Stefik 82b].

Staff: D. Bobrow (Xerox), M. Stefik (Xerox).

Some of the project programming has utilized the GLISP compiler system [Novak 82]. During the past six months GLISP has been extended and made more robust, and a graphics editor based on GLISP object descriptions has been developed for the Xerox D1100 Workstation (the Dolphin).

The GLISP language has been extended as an experimental hardware description language. This language allows hardware data structures such as computer instruction formats to be described and used in a natural way. The GLISP descriptions are compiled into an intermediate code which is similar to exisiting register tranfer languages. This intermediate code runs on an interactive simulator using the Dolphin graphics system.

Staff: G. Novak.

1.5 Electrically based layout system, Lava

Lava is an electrically based, general-purpose layout language. Our principal objectives are topological, rather than geometric, layout description and guaranteed design-rule correctness of layouts. Lava's major components are a sticks compactor, cell stretching and abutment mechanisms, a router, and a framework to link them together.

We have rewritten Lava to stabilize it and to incorporate some of the hooks that will be necessary for further investigations. We have concentrated on a clean implementation of the aspects that we understand well, removing some of the more ill-conceived mechanisms in the previous implementation. One major improvement is that much of the technology-specific information is now centralized; while Lava is not intended to be technology-independent, this technology file makes it possible to change easily parameters of the nMOS target process.

The result is a sufficiently stable platform for further investigations, for example, a well-conceived composition level and logic-to-sticks conversion. The rewritten Lava now successfully compiles a large number of test cases, including (very recently) the serial memory chip described in our previous report.

Staff: C. Burns, D. Chapiro, P. Eichenberger, R. Mathews, J. Newkirk, D. Perkins, T. Saxe

Related Efforts: EARL (CalTech), CABBAGE (UCB)

1.6 Routing

We have developed a new, 2-dimensional area router, the loop routing scheme (LRS). LRS handles both rectangular- and doughnut- shaped routing areas. LRS is a promising box router for the custom routing problem because, like the dogleg channel router, it indicates how much expansion of the routing area is necessary to complete a given routing.

The difficulty of channel routing problems, and the performance of channel routers, may be measured by the number of wiring tracks required to complete the routing. Previously, no similar measures existed for comparing area routers. Such a measure must describe how difficult a given, fixed, area-routing problem is, since there is no well-defined way to expand the routing area to guarantee completion of the route.

We have developed an appropriate measure of problem difficulty, the Manhattan Area Measure. By using it to assess the difficulty of routing problems generated using Monte Carlo techniques, we have compared the performance of LRS to the classic Lee area-routing algorithm. The LRS has vastly superior performance to the Lee, successfully routing problems that are twice as dense as those that the Lee will complete successfully.

Staff: T. Saxe, L. Smith

Related Efforts: PI project (MIT)

References: [Smith 82]

1.7 Logic-to-sticks conversion, Dumbo

This work is aimed at simplifying the layout of random logic. Some amount of glue is inevitable in a design, but is is painful to lay out and typically does not consume a significant amount of the total area of the design. Consequently, we are searching for techniques for converting logic, specified as transistors, gates, and a net list, to stick diagrams for compaction by Lava.

The logic-to-sticks conversion program Dumbo has now produced some layouts of small cells with respectably small areas completely automatically. For example, a 12-transistor cell was laid out with no area penalty when compared to the original hand-drawn sticks. Force-directed placement and orientation of components made this result possible. However, for larger cells we still see penalties of 100-150%.

We are continuing to analyze the sources of inefficiency in Dumbo layouts. As for our custom-chip router described in our previous report, a series of small 10% efficiencies cumulate to create a large overall area penalty. We are analyzing these sources of error to try to understand and control them. However, one major source of inefficiency appears to be the sensitivity of our sticks compactor to small changes in the stick diagram. While the human designer copes with these vagaries very well, we do not understand them well enough to permit us to avoid them in automatically generated cells. Nevertheless, we feel we will be able to make substantial additional progress in this area.

Staff: R. Mathews, D. Perkins, W. Wolf

Related Efforts: Rule-based circuits-to-sticks conversion (A. Bell, PARC)

References: [Wolf 82]

2 VLSI Processor Architecture

2.1 MIPS - A High-Speed Single-Chip VLSI Processor

MIPS (Microprocessor without Interlock between Pipe Stages) is a project to develop a high speed (> 1 MIP) single-chip 32-bit microprocessor. Like the RISC project at Berkeley, MIPS uses a simplified instruction set and is a load-store architecture.

The MIPS architecture is summarized in a previous technical progress report and is discussed in several publications. During this six month period, the major goal of the project was to develop a series of test chips that would provide a complete test of each major component of the chip individually.

The six test chips contain all the parts needed to implement the complete MIPS processor. Each test chip also contains additional testing and pin multiplexing hardware. By fully testing the components before fabricating a complete design, the probability of success on the first run is much higher. This approach also allows us to characterize the indivdual components and make performance adjustments before the final fabrication. By designing a single reusable test frame, the individual test chips may be constructed from the exact pieces of the complete chip with a minimal amount of effort. The final assembly process consists of merely composing the individual test components to form the complete processor. Lastly, this process offers an ideal opportunity to test the concept of fast-turnaround foundries. Because progress on the project depends on receiving and testing the chips prior to completing the final design, reasonable quality, fast-turnaround fabriction is essential.

The six test chips and their current status is as follows:

1. Register File Test Chip - submitted and tested at both 3 and 4 microns. The 4 micron chips were functional, although the yield was <10% (i.e. all parts of the chip worked over 10 die, but no single die was completely functional). The 3υ fabrication produced no working parts.

2. Instruction Decode Test Chip - tested one out of eight die was completely functional.

3. Barrel Shifter Test Chip - preliminary tests from the first fabrication have shown partially working chips. So far, no definitive problems have been identified.

4. ALU Test Chip - destined for November submission. Currently running functional simulation.

5. Program Counter Test Chip - version 0 currently in test. Version 1 in simulation.

6. Master Pipeline Control Test Chip - submitted in October fabrication run. Not returned yet.

Staff: F. Baskett, J. Burnett, J. Gill, K. Gopinath, T. Gross, J. Hennessy, N. Jouppi, W. Park, S. Przybylski, C. Rowen, A. Strong.

Related Efforts: RISC (UCB), IBM 801 (IBM Yorktown), Cray-II (Cray Research).

References: [HennessyJouppiPrzybylski 82, Gross 82, Hennessy 83]

2.2 Geometry Engine

The Geometry Engine is a high-performance, floating-point computing engine for geometric operations in 2D and 3D computer graphics. Multiple copies of the Geometry Engine provide a parallel computing system with very high-performance. (5-10 million floating-point operations per second.)

During this period, we obtained a second fabrication of Geometry Engines. This batch provided enough chips to build a geometry system (10 chips) and a complete prototype. This prototype system, called the IRIS, is discussed in the next section. The Geometry Engine design is completely functional, although the performance is less than originally predicted.

Staff: J. Clark, M. Hannah

References: [Clark 82]

2.3 IRIS Workstation

The goal of the IRIS workstation project was to design a high-resolution, color, extra high-performance graphics workstation that utilized all of the features of the Geometry Engine and was software-compatible with the SUN 68000 processor (excluding graphics software compatibility). The system consists of

- A SUN-compatible processor/memory board.
- A Geometry Engine board (10 Geometry Engines).
- A Raster Generation Subsystem.
- A Raster Update Subsystem.

The IRIS allows the user program to generate display programs that provide for real-time motion of 2D and 3D environments, multi-window displays and color lookup table manipulation. To provide for motion simulation, the system is dynamically configurable to provide either double or single-buffer images. The system has been in operation since August, 1982, and procedures are underway to replicate copies of the system for future research at various Stanford Laboratories.

Staff: K. Akeley, J. Clark, M. Grossman, C. Rhodes

3 Signal Processing Algorithms and Architectures

3.1 Simulation of Musical Instruments

The Digitar chip uses a digital-filter method to synthesize various waveforms. It consists of about 3500 transistors, and was designed by writing a SAIL program to generate its layout. Internally, it has a 12-bit arithmetic unit that adds, subtracts, increments, decrements, and single-bit shifts; it has a 16 by 15-bit shift register array and a 7 by 21-bit microcode array for controlling the data path. Although primarily designed to synthesize plucked string sounds, the chip can also produce snare-drum, clarinet, and bassoon timbres. It is controlled by an 8-bit port and internally decodes the command sequences sent to it by a microprocessor. The chip requires 4K of external RAM in order to perform its functions, and it addresses this RAM directly. Support for this project was primarily by the Hertz Foundation; ARPA support has been limited to computation facilities and a small amount of personnel support.

Staff: K. Karplus, A. Strong

4 Testing

4.1 Process Control Test Structures

Work continues on the development of a variety of test structures for use in providing feedback to the wafer fabrication activities at Stanford. The development of these test structures has paralleled the re-establishment of the wafer fabrication activities as well as the tighter process control requirements imposed by the development of a 2 μm CMOS.

4.1.1 Defect Density Test Structures

A set of test structures has been developed which allows the extraction of defect densities associated with step-coverage, contact opening, and shorts which may be encountered at various points in the fabrication sequence. Although meander patterns, contact strings, etc. provide a means of generating go/no-go statistics of these failures for a large coverage area, they provide little data as to where the defect actually occurs or even if a test failed because of one or many defects. The advantage of these types of tests, however, is the fact that they cover many sample sites and a large silicon area with a single measurement. The test structures that we are developing attempt to provide a greater degree of localization to better pinpoint the number and location of defects without totally sacrificing the ability to sample numbers of sample sites. These devices provide a means of addressing the test stucture, if desired, to help localize either the position or number of defects. The addressing circuitry allows a number of short test sections to be abutted together in a serial fashion to provide go/no-go testing over the full array. If the array fails, the short sections can be addressed either individually or in clusters using a binary search to progressively eliminate fault-free sections. The addressing circuitry is, of course, provided with a self-test mode to insure that we do not misinterpret peripheral failures as being due to the cell array. Initial designs have divided the array into only 16 sections to limit the number of pins required for address electronics so that we may maintain full compatability with the NBS 2 by 10 probe array.

4.1.2 2 Micron CMOS Test Structures

A full set of test devices has been developed to characterize the 2 μm CMOS process. These test structures attempt to study the performance of the n- and p-channel devices (both individually and taken together) as well as a number of the important parasitic effects which are increasingly important as the feature size decreases.

Even though CMOS is largely a ratioless technology (although it will not perform optimally with grossly mismatched Z/L ratios), we wish to examine inverter performance to pick the optimum size for both the n-channel and the p-channel devices. At larger feature sizes (~ 4 μm and greater), the p-channel device is often fabricated with a larger Z/L ratio than the n-channel device to offset the fact the electron mobility is greater than hole mobility at comparable doping levels. At a two micron feature size, this difference is not as large because (a) the n-channel device is more severely dominated by velocity saturation effects than the p-channel device and (b) for equivalent *drawn* gate lengths, the p-channel device will usually have a shorter effective channel length because of increased lateral penetration of the source/drain regions.

Latch-up is the parasitic device phenomenon in CMOS which deserves the closest scrutiny and good test structures are essential to the characterization of the latch-resistance of any CMOS technology. Because the operating voltages in small geometry integrated circuits have not been reduced, hot carriers are much more likely to initiate latch-up in CMOS. For these reasons, our test structures include a variety of devices which will be used to study the latch-up behaviour and hot carrier effects in 2 μm CMOS technology

4.1.3 Measurement Hardware

Our parametric measurement capabilities have been extensively modified during recent months. A HP 4145 Semiconductor Measurement System and a HP 6942 matrix switch have been added to our process/device characterization system. A HP 9845B controls these instruments as well as a Rucker and Kolls 1032 wafer prober and allows independent specification of the wafer stepping pattern, the matrix connections to the probe card, and the actual test to be performed. A preliminary investigation indicates that this software will be readily compatible with a HP9826/36, should the need arise. We now have improved software which drives the R & K 1032, so that our probing speed has been roughly doubled in recent months.

Staff: T. Walker, L. Gerzberg, W. Yarbrough

Related Efforts: M. Buehler (JPL), L. Linholm (NBS), V. Tyree (MOSIS), D. Trotter (Miss. St.)

4.2 The ICTEST System

The ICTEST system is a unified system for functional simulation and testing. The test is written in ICTEST, a superset of C extended to include testing primitives, data formatting, and mechanisms for specifying parallelism and pipelining. The test may then be targeted to run against a simulator (ESIM or TSIM) or a tester (MINIMAL, MEDIUM, or TEK S-3260). The MEDIUM tester is the testing workhorse; the TEK tester is intended primarily for performance measurement and functional testing at speed.

ICTEST itself has remained relatively stable over this period. We continue to use it to test our designs, including the MIPS test chips. Support for the clocking discipline is now substantially debugged, although we need to rethink our approach to qualified clocks and decide how they might be supported on the TEK (if that is indeed possible).

We are reducing the MEDIUM tester to a chip set. It will connect to a standard DEC DMA interface, and we shall distribute it to the community when it becomes available. The chip set that we have designed comprises 2 chip types, and a 64 pin tester will require a total of 3 chips. The tester control and pin electronic chips have both been designed and submitted for fabrication. Additionally, we have designed test chips for some new circuits that we need, including pads capable of driving 30mA loads. We have received the pin electronics chip and the pad test chip and partially tested them. They both seem to work correctly.

Staff: D. Boyle, D. Marple, R. Mathews, J. Newkirk, I. Watson

Related Work: FIFI Project (CalTech)

References: [Mathews 82], [Watson 82]

4.3 Clocking Discipline

We have developed a 2-phase clocking notation and an associated clocking discipline. The objective is to provide appropriate formal concepts for thinking about clocking in 2-phase systems, and to delineate a circuit syntax guaranteeing consistent clocking. The clocking discipline can also be co-opted to guarantee other forms of correctness, *e.g.*, freedom from charge sharing. The auditing tool *clockck* checks circuits extracted by the ESIM extractor for conformance to the discipline.

We have finished gleaning information from the Winter '82 testing class. Of 8 chips fabricated and tested, 3 designs had fatal clocking errors, and 2 had errors that could hurt performance. One of the three fatal cases was of the most interesting sort: a design that passed extensive simulation completely, but failed the clocking check and did not, in fact, work when fabicated.

Staff: R. Mathews, J. Newkirk, D. Noice

Related Efforts: Glasser's work (MIT)

References: [Noice 82]

4.4 Practical Testing

We have tested 30 more copies of a 10,000-transistor serial memory based on a 3-transistor RAM cell. The memory was originally intended as a step toward a serial signal processing system, but has actually proved to be test of our testing system and our understanding of the technology.

Of the 30 new parts from run M1DV, 40% are defect-free. Sixteen percent show anomalously low (less than 100-microsecond) storage times of the sort we reported previously. The remainder have failures that may be explained in terms of fabrication defects, with the exception of 2 chips that have the mysterious property that while every bit in the memory plane seems to be functioning correctly, when we apply error correction to this perfect data, errors result!

As a result of our frustration with short storage times, we have designed and tested a canary circuit that monitors storage times directly. The storage-time oscillator is a 3-stage ring oscillator, one stage of which is a storage node that is charged and allowed to relax toward ground.

We have tested 2 chips so far, one with very short (1 microsecond) storage times. The storage oscillator successfully indicates that the short-storage time chip is defective and that its companion is acceptable. Using optical injection to vary storage times over a large range, we have found that the storage oscillator on each chip predicts the storage time very precisely. The design seems to be insensitive to power supply variations, but we will need to test more chips before we are completely confident.

As a simple experiment in performance measurement, we have designed and tested an instrumented family of 7 PLAs with different loading characteristics. We have measured the performance of each path through each chip and computed regression lines to fit the observed data with delays predicted by τ models. The observed fits are very good, with correlation coefficients around .8 and derived τ's ranging from .25 to .57 nanoseconds. However, the intercepts of the regression line are non-zero, indicating systematic measurement errors specific to each member of the family. We are currently trying to chase these errors to ground.

Staff: G. Eckert, R. Mathews, J. Newkirk, T. Saxe, L. Shwetz, I. Watson

References: [Saxe 82]

5 Theoretical Investigations

5.1 Connected Components Algorithms

Finding the connected components of a graph, given its adjacency matrix, is a problem that has received much attention recently, but the best way to implement the algorithm in VLSI is not known. Using the AT^2 measure, it is possible to solve the problem in $n^{2+\epsilon}$ for an n node graph if one uses the mesh-of-trees, a layout that requires a great deal of area. Lipton and Valdes considered layouts that used area proportional to the number of nodes, and came up with an $n^{3+\epsilon}$ algorithm that is, unfortunately, not when-oblivious; the time at which inputs are required depends on the data. A. Siegel has recently invented an algorithm that runs in the same time as the Lipton-Valdes algorithm, is when-oblivious, and uses only n pads. The area is on the order of $n^{3/2}$, so its figure of merit is $AT^2 \approx n^{3.5+\epsilon}$, which is better than any known n-pad algorithm. The material has not yet been written down by the author, but a sketch appears in [Ullman 82b].

Staff: A. Siegel, J. Ullman

5.2 Modular Model of Event-based Concurrent Systems

The formal model we have been developing has two major components: a structural algebra for describing module interconnection structures, and a behavioral semantics that defines the function computed by a network of modules. Most of our work in the last six months has concentrated on the behavioral semantics.

As described in a previous report, the behavioral semantics associates with each module and network of modules:

- a functional mapping between partially ordered events at input and output ports,
- a domain constraint, specifying that certain output events must precede certain input events, and
- a functional constraint, specifying that certain input events must precede certain output events.

The domain constraint is essentially a statement of the conditions under which the module can be expected to work correctly. For example, it might require that no new input events arrive until after all outputs for the current input values have been produced. If the domain constraint is violated, the behavior becomes unpredicatable. The functional constraint, on the other hand, contains information about when a module will produce new output events. Thus the domain constraint tells what the module requires of its environment, and the functional constraint tells what it guarantees.

Our recent work has been particularly concerned with the problems of module substitution and the semantics of non-deterministic systems. The module substitution issue arises because we often wish to substitute one module for another in a network and need to know when this can be done without affecting the properties of the network. A simple criterion for such substitution is *semantic equivalence.* If two modules have the same functional mapping, domain constraint, and functional constraint, then one may replace the other without any change in the network's behavior.

In some cases, however, we need a more flexible criterion. We would like to be able to make a substitution so long as it allows the network to continue working correctly and produce the same output. This may be possible even with modules that are not identical. For example, suppose we have a system containing a module that can perform correctly as long as it is asked to buffer no more than three input elements at a time. (This would be expressed in the domain constraint.) If we replace this module with one that is identical except for the fact that it can buffer more items, then the new network should continue to work correctly. In general, we can always replace a module by one with a *weaker* domain constraint. If the environment of the original module guaranteed that the stronger domain constraint was satisfied, then the weaker one will necessarily be satisfied, and the composed system will continue to perform correctly. Likewise, we can always substitute a module with a *stronger* functional constraint, because if the original module operated in a way that satisfied the domain constraints of other parts of the network, then the (more constrained) new module must do so too. Thus we can perform such a substitution if the new module has an identical functional mapping, weaker (or identical) domain constraints, and stronger (or identical) functional constraints. The new network may not be equivalent to the old, but it will operate correctly in any environment where the old one does. This sort of substitution arises very naturally when the original system is viewed as a *specification* and the substitution represents an *implementation* of the specification.

The second problem we have considered is extending the semantics to non-deterministic systems. Non-determinism is a property of many concurrent systems. It may arise even in networks where all the primitive modules are deterministic; this is because the relative timing of events at different modules is unpredictable, and different timings may cause the system to produce different outputs. A non-deterministic module can be described by altering the functional mapping to give, for each input, a *set* of possible outputs. There are several technical problems that must be resolved in this sort of definition. The most significant is being able to guarantee that loop-feedback (the operation that sends some of a network's output to its own input ports) is always well-defined. By modifying the approach of Plotkin and Smyth, which deals with non-determinism in state-oriented rather than event-oriented models, we have been able to solve these problems and develop a mathematically sound semantics for non-deterministic networks.

Staff: S. Owicki and N. Yamanouchi

5.3 Defect Tolerance in Array Architectures

We have developed a new body of theory treating the effect that defects have on yield of array architectures. The theory addresses such issues as whether it is possible to find chains or arrays of working elements embedded in a large array and what reconfiguration capabilities must be available for the yield of the reconfiguration process to be non-zero.

For the problem of finding a connected chain of working elements in a square array, we have developed a new algorithm that requires time linear in the number of elements to be chained. We have also made progress on the problem of finding an array of working elements embedded within a larger array by tightening the bounds determining when such reconfiguration is possible. We are beginning some new work investigating the effects of defects in the interconnect itself.

Staff: A. El Gamal, J. Greene

References: [GreeneEl Gamal 82]

5.4 Wiring Area for Gate Arrays

By applying statistical modeling techniques, we have developed a body of theory that predicts how to realize a given function in a gate array with smallest overall die size. The central result is that it is preferable to have a smaller gate array with a larger number of tracks in between blocks, thereby permitting higher overall use of the array elements, rather than sparsely using a larger array. These theoretical results are borne out by a large body of empirical data collected by IBM.

Staff: A. El Gamal

References: [EL Gamal 82]

6 The SUN Workstation: File System Development

The SUN represents a radical departure from the customary workstation design in that it does not have a local disk. A typical SUN workstation at Stanford has 256K bytes of memory, a frame buffer with some additional memory to hold the raster image, and a high-performance Ethernet link. In order to use the SUN as a workstation for VLSI design or other applications, it must be possible to read and write file storage from software resident in the SUN.

Our initial approach, which permits us to run simple software on the SUN, was to implement a page-at-a-time file server called a Leaf Server, which ran on a supporting computer (usually a Vax) and provided disk page access in response to request packets. This server and its development were reported in last year's progress report, and we have done little work to it since. It is worth mentioning that the Unix-based Leaf Server that we wrote last year was made available to other Arpa-supported users of Xerox 1100 Lisp Workstations, and (after suitable modifications at ISI to make it compatible with the Xerox equipment) it provides a valuable disk support facility to AI programmers using the 1100's.

Our experience with the Leaf Server approach to Remote File Access demonstrated conclusively that a single-host file server was not an adequate level of file support for a network machine participating in a distributed system. We experimented for a few months with changes that could be made to the Unix file system or to the behavior of the Unix Leaf Servers that would make a more reasonable distributed file system available to SUN users. We abandoned this approach for three reasons:

- The Unix file system does not map neatly to a distributed environment. At the design level, it assumes that there is at most one copy of any file, and that the entire file system is tree-structured. It is difficult to modify the Unix kernel to think that parts of its file systems are on other machines, though at the 1981 SIGOPS conference some Bell Labs researchers reported having accomplished it (with a severe degradation in performance.) At the implementation level, its locking mechanisms are unreliable and the fixed I/O table sizes in the kernel provide unreasonable fixed bounds on the total number of files that can be accessed simultaneously on a given server host. We thought that we could get by with a combination of a few kludged Unix file systems for the first generation of SUN software, but the problems overwhelmed us.

- Leaf-Server access to files on a time-shared Unix system was a second class citizen. We frequently found the need to log a job on to the host Unix, probe around the file system, then log off and resume a stopped SUN job that was having file problems. This problem could be bandaged by providing a set of file utility programs resident on the SUN, but the essence of the problem is that a Unix file system is not very suitable for a non-Unix-like operating system; we do not intend to run Unix on our SUNs.

- Even with just 4 server hosts available (3 Unix and one Tenex), the amount of context that needed to be maintained in a user's head was overwhelming. The lack of automatic location, migration, or replication facilities made it particularly difficult to find files whose precise location was not known.

We therefore, reluctantly, concluded that we were going to have to design and implement our own file system to our own specifications. This file system would be network-wide, provide a uniform set of access mechanisms and management tools, and be implemented on a variety of file computers.

We have settled on a multi-level design based around a central archival file system and distributed cached copies. We are implementing the design "from the inside out", starting with the reliable archival part and working towards the fast cache servers; the reasoning behind that decision being that it is better to have a slow reliable file system than a fast unreliable file system during the development phase. We intend to present a paper on this system at the 1983 SIGOPS conference, whose paper submittal deadline is in January, and we intend to have the archival server working and the migration protocols designed by that time. The initial implementation is taking place on our time-shared VAX, but we hope to move to a dedicated machine with a larger disk as soon as it becomes available.

Staff: J. Mogul, B. Reid

References: [Baskett 82]

6.1 Computer support for a Fast Turnaround Laboratory

We have completed the planning and exploratory stages of a project to provide extensive automation and computer support for the Fast Turnaround Laboratory. This ambitious interdisciplinary project (involving researchers from Computer Systems, Integrated Circuits, and Solid State laboratories) will provide control, documentation, training, portability, repeatability, and efficiency in the area of IC fabrication processes.

As a result of this exploration, we have isolated the following goals for this project:

- Automatic control of the IC fabrication processing equipment.
- Integration of fabrication control with simulation control, to run simulation and fabrication in parallel.
- A transportable, repeatable means for recording a fabrication process.
- The ability to repeat an arbitrary process on demand, for demand production of parts.
- The ability to manage and schedule a single IC fabrication line for multiple purposes.

- Computer aids for training and documentation.
- General data processing and database support for analytic work in the Laboratories.

We recognize that particular technical achievements will contribute to the realization of several of these goals. The most important of these is the development of a language for the representation of fabrication processes. When this language exists, it can be used as input to the control system, as input to the simulation system, as the contents of an archive for demand production of parts, and as a basis for training and documentation aids. Furthermore, the nature of this language will color most of the other work.

We have therefore spent several months on the preliminary design of such a language, our Language for Specifying Manufacturing Processes, or LSMP. (We intend to change its name before its specification is published, but we are for the moment calling it LSMP internally). This language resembles Ada semantically, but contains built-in type support for non-numeric types pertaining to manufacturing, and contains extension and package mechanisms suitable for the description of typed objects whose physical existence is outside the controlling computer, e.g. furnaces. We quickly found that two languages were necessary, one to describe the manufacturing process and another to describe the fabrication line itself. This second language corresponds very much to the microcode found on conventional computers, and it is used to implement a somewhat abstract instruction set, into which the LSMP is compiled. A compiled LSMP program can operate an automated fabrication line with the appropriate microcode, or it can operate a simulation system (one component of which would be programs like SUPREM) with a different set of microcode. Other microcode would be written to permit experimentation and testing of new processes before actually turning them loose on the fabrication equipment.

Only by implementing all pieces of this system and actually using it to manufacture parts can we satisfy ourselves that it is complete; we would therefore want to do an automating implementation even if that were not one of the goals of the project.

Having completed the first level of implementation, we intend to write software that amounts to a distributed time-shared operating system for the fabrication line; this will permit multiple independent fabrication processes, or laboratory experimentations, to be run on the same fabrication line simultaneously just as a time-shared computer is now capable of running independent programs simultaneously. This operating system will also take responsibility for the long-term scheduling and priority realization.

We consider that a system like this will be a superb testbed for explorations in knowledge-based systems for training and diagnosis, and for applications of interactive graphics, computer aided instruction, and reliable models of computation. We therefore hope to attract talented graduate students from various areas related to computer science, in addition to the core of people knowledgeable in IC fabrication, to this project. Currently Prof. Brian Reid is spending 90% of his research time on this project, and one graduate student, Harold Ossher, is working on it full time. Several more graduate students are eager to join as soon as funding becomes available to them.

7 Fast Turn-Around Laboratory

Activities if the Fast Turn-Around Laboratory have concentrated on the characterization of a significant amount of fabrication, mask making, and testing equipment followed by the subsequent incorporation of these items into the primary NMOS and CMOS processes. The following subsections of this report will detail the performance of many of these pieces of equipment and indicate how they have enabled us to increase the quality of our NMOS/CMOS wafer production and the quality of our device research.

7.1 Wafer Fabrication

During this period upgrading the processing lab has continued, in order to meet the project goals of establishing standard 2 micron CMOS and NMOS processes. Much of this effort was directed at bringing on-line and characterizing the equipment ordered during the previous reporting period. This equipment includes three low pressure chemical vapor deposition (LPCVD) systems, two plasma dry etchers, and a photoresist processing system. In addition, a 1:1 projection alignment system capable of 1.25 micron linewidths was installed and is being characterized. Finally a sputtering system and linewidth measurement system have been ordered.

7.1.1 Low-Pressure Chemical Vapor Deposition

The older multi-purpose atmospheric Epi/CVD system has been replaced by three (poly, nitride and oxide) dedicated LPCVD sytems. Low pressure deposition offers significant improvements over atmospheric deposition as following:

1. Improved thickness uniformity (2% vs. 15%) due to the 10^3 improvement in the gaseous diffusion coefficients.

2. Fewer particle generated defects due to vertical wafer positioning and the "hot" wall deposition on the tube in the LPCVD system, versus horizontal positioning and "cold" wall deposition in the atmospheric system.

Nitride CVD is essential to the local oxidation isolation process used for NMOS and CMOS. Use of the LPCVD nitride system is now standard in our lab, and is capable of depositing the required 800 Angstrom film to a uniformity of 2%. The typical deposition rate is 36 Angstrom/min. The gas flows are 20 SCCM of dichlorosilane and of 60 SCCM of ammonia. The furnace temperature is 790 degrees Celsius and the deposition pressure is 500 mT.

Doped polysilicon is used as the gate material and as a conductor in both NMOS and CMOS. The change over to LPCVD not only improved thickness uniformity and particle control, but also reduces the grain size because of the lower deposition temperature (620 degrees C for LPCVD vs. 900 degrees C for APCVD). The smaller grain size is important since it reduces grain boundary related oxide defects, and reduces poly edge roughness. The poly deposition is also done at a pressure of 500 mT and with a silane flow of 30 SCCM. The deposition rate is 90 Angstrom/min.

The last of the new LPCVD systems which is operational is the low temperature oxide (LTO) system. This phosphorus-doped oxide deposited at 450?C is intended to replace the atmosphere-deposited vapox layer which is reflowed to improve step coverage. Initial undoped LTO films show excellent uniformly (roughly 1%) and conformal step coverage. The deposition pressure is 400 mT and the flows are 60 SCCM of silane and 95 SCCM of oxygen. Phosphorus doping during deposition is now being characterized.

7.1.2 Plasma Etching

In order to achieve line widths of three microns and less, wet etching must be replaced with dry etching techniques. For poly and nitride etching, a Drytek RIE 100 etcher has been purchased. This etcher,which is in routine operation, operates in the plasma etch mode. It is fully automated to eliminate handling-induced defects and to give better process control. A key feature is its use of a interferametric laser end point detection system. For poly etching, a controlled slope process has been achieved which gives a 70 degree wall slope with a critical dimension loss of only 0.1 micron per edge. Controlled slope, as opposed to pure 90 anisotropic etching, is desirable for step coverage needs. The present process uses a mixture of C_2ClF_5 and SF_6 both at flows of 50 SCCM. The pressure is controlled at 150 mT and the RF power density is at 0.3 watts per cm^2. For this process, the etch rate is 2000 Angstrom/min, and the selectivity of poly to both the resist and the underlying oxide is 20:1.

For the less critical etching of silicon nitride films, an isotropic process is used. This process uses a mixture of CF_4 and O_2 with flows of 90 and 10 SCCM, repsectively. The pressure is again set at 150 mT and the RF power is again 0.3 watts per cm^2. The etch rate is 160 Angstrom/min with selectivity to oxide of 4 to 1.

For the etching of SiO_2, Branson/IPC has given us one of their Sigma 80 etchers. This unit is an automated single-wafer-at-a-time machine. The SiO_2 process uses a mixture of C_2F_6, CHF_3 and He at respective flows of 300, 300, and 3000 SCCM. The pressure used is 10 Torr and the power density is 5 watt/ cm_2. The etch rate on thermal oxide is 5000 Angstrom/min and the selective to both resist and silicon is roughly 7:1. The resulting oxide wall slope is currently 75 degrees. The process is being modified to improve selectivity, uniformity, and wall slope.

7.1.3 Photolithography

An automated photoresist resist processor has been brought on line. This machine offers full cassette operation with microsprocessor control of all functions and will significantly reduce resist related defects. Functions include priming, resist coating, "puddle" development, and microwave baking. The processor is in standard use except for the microwave bake feature, which will replace the use of the resist bake oven when characterized.

An Ultratech 1:1 projection stepper has recently been installed and is being characterized. This machine offers significant area utilization improvements over our Canon 4:1 manual stepper, which is limited to a *total* exposed area of only 3 cm by 3 cm. The Ultratech is a fully automated state-of-the-art optical alignment system which has auto focus, alignment, and load. It has a working resolution of 1.25 microns, with an alignment accuracy of 0.14 microns. The pellicle protected reticle has four selectable fields which offer a maximum *unique* silicon area of 6 cm^2. An advantage of this stepper over the popular 10:1 systems is that it uses two wavelengths (405 and 436 nm) and thus is less susceptable to standing wave problems.

An optical linewidth measurement system has been ordered. This auto focussing unit will be used to obtain tighter control of linewidths during processing.

These pieces of fabrication equipment have been used in the fabrication of both NMOS and CMOS device wafers. The performance of the CMOS devices will be discussed in the Device Research subsection.

7.2 Electron Beam Lithography

The principal activity of the E-beam lithography group involved the installation, characterization, and, finally, the on-site acceptance of the Perkin-Elmer/ETEC MEBES machine. The on-site acceptance tests included extensive testing of the registration (alignment) accuracy of the MEBES machine in anticipation of its use as a direct-write lithography tool. During the period when the E-beam machine was undergoing acceptance tests, we were characterizing the resist developing and chrome etching systems in our lab. The first use of the ability to develop and etch chrome plates was to produce the 2 μm CMOS test plates for Jim Pfiester. These plates were actually written at P-E/ETEC, but developed and etched at Stanford, before our machine had passed on-site acceptance.

At present, we are in the process of making masks for the Ultratech 900 projection lithography system. The

Ultratech is a 1:1 wafer stepper whose mask requirements are somewhat different from that of a Perkin-Elmer 140/240, so we are at present carefully formatting these plates manually rather than using the MOSIS service. It does appear, however, that we will be able to automatically place the desired dice, alignment marks, and scribe lanes. This first mask set will include Jim Clark's Geometry Engine, several of John Hennessy's MIPS test vehicles, and the two chips designed by Newkirk/Mathews/Watson that comprise a medium tester.

In addition to working with the MEBES machine, we have been developing a tri-level resist technology for use in direct-write applications. The principal need for a multi-layer direct-write technology is to provide a thick, chemically resistant "working" layer of resist on the wafer and yet maintain a thin, high resolution layer of resist for the actual electron patterning.

In our tri-level, the underlying layer of resist is 1.2 μm of AZ-1470 which has been baked at 200 degrees Celcius to remove any photosensitivity. The AZ-1470 has very good chemical resistance, planarizes the wafer surface, and reduces secondary electron backscatter (compared to a silicon substrate) which can be a source of image degradation. The second layer is a thin (500-800 Angstrom) layer of poly-Si which will ultimately be used as the intermediate masking layer in the process of transferring the pattern from the PBS electron resist to the underlying AZ 1470. We wish to keep this layer thin and of low atomic mass to minimize secondary electron image degradation. We have used both evaporation and plasma-enhanced chemical vapor deposition to deposit this layer. Researchers who are exploring tri-level resist for use in optical lithography favor the use of SiO_2 as the intermediate material because of its low index of refraction. For electron exposure, however, the slight conductivity of poly-Si is preferable to reduce charging effects. The top layer is 0.4 μm of PBS electron resist which is coated, exposed, and developed as if on a chrome blank. This pattern is then transferred to the poly-Si using plasma etching (an anisotropic CF_4 + 4% O_2 etch is adequate because the poly-Si is so thin) which in turn serves as a mask for O_2 reactive ion etching. Using a partial pressure of 6 μm Hg of O_2, we have achieved nearly vertical sidewalls in features 0.5 μm wide separated by 0.5 μm.

We have been investigating two reactive ion etching systems for routine use in this application: one is manufacturered by Materials Research Corporation and the other is manufacturered by Drytek.

7.3 Device Research

Many different aspects of our device research program have an impact on the development of a 2μm CMOS technology. Jim Pfiester has designed a mask set containing a wide variety of CMOS test structures to aid him in the development of this process. The mask set was generated without the use of bloats or shrinks during mask making to provide an accurate measure of what bloats/shrinks will be required to produce the "drawn" dimension in silicon. This is an n-well process which has a surface concentration that does not require a channel stop for the p-channel devices. The n-channel drain/source junction depths are only 0.3 μm deep and the p-channel drain/source region are about 0.55 μm deep. The measured electrical channel length for the n-channel and p-channel devices was 1.6 and 1.1 μm for a *drawn* dimension of 2.0 μm, indicating that our poly-Si plasma etching is providing an anisotropic etch profile because most of the difference between the effective and drawn channel lenghts is due to the lateral diffuaion from the drain and source regions.

Electrically, the n-channel devices look very good in terms of leakage current, threshold voltage. Previous runs had indicated that drain-induced barrier lowering (DIBL) is the most stringent test of these devices. These devices exhibit very good drain-induced barrier lowering properties. The threshold voltage of the p-channel devices are as we expect them to be. Unfortunately, these first devices had a parasitic leakage current from drain to substrate along the surface of the n-well which is superimposed on the actual drain/source current. The cause of this leakage is under close scrutiny and two sets of additional CMOS wafers are nearing the end of the fabrication sequence.

In order to help in the investigation of latch-up in short channel CMOS structures, we have initiated a program to investigate the properties of devices as a function of temperature. From a device physics standpoint, it is extremely desirable to have such a capability in order to establish the activation energy of the phenomena under investigation. We are at present constructing a test fixture which will allow us to scan the temperature of a packaged device from 77 degess Kelvin to 100 degrees Celsius.

Staff: J. Shott, J. McVittie, E. Wood, K Saraswat, R. Castellano, F. Pease, D. Dameron, C.-C. Fu, P. Jerabek, J. Plummer, J. Pfiester, T. Nguyen, L. Lewyn, J. Marshall, A. Henning, D. Gardner

8 Other Projects

8.1 Polygon Package and Design-Rule Checker

For some time now, we have made available a high-quality design-rule checker based on our polygon package. It derives circuit connectivity information to prevent reporting of false separation errors between electrically connected components. This checker is used by our design classes and for our research and has been heavily tested by 80+ designers. We have recently added support for buried contacts in nMOS.

We have developed and tested an analogous checker for the JPL bulk CMOS rules. It has checked the designs submitted by Stanford, MIT, and Lincoln Labs for the bulk CMOS run. We have distributed the CMOS checker to JPL and MIT.

Staff: D. Noice

8.2 Cell Library

The nMOS cell library is now being prepared for publication by Addison-Wesley as a companion to the Mead and Conway text. Accordingly, we are cleaning up the documentation, correcting minor design-rule violations, and thoroughly checking the cells. Some new cells will be included, such as LSSD PLA buffers.

We have designed and submitted bulk CMOS cells to form the basis of a CMOS cell library. The new cells are pads, PLA designs, and counters.

Staff: R. Mathews, J. Newkirk, J. Shott, T. Walker

8.3 Modifications to MIT Circuit Extractor

We have integrated the MIT circuit extractor with our CLL/CIF processing software, resulting in an order-of-magnitude improvement in extraction speed. Previously, the 10,000-transistor serial memory required several hours to extract; extraction now requires approximately 10 minutes. We will distribute the extractor if there is sufficient interest; however, prospective users should be aware that our CIF processing system is restricted to Manhattan-only, rectangle-only designs.

Staff: J. Newkirk, T. Saxe, S. Taylor

References

[Baskett 82] Bechtolsheim, A., Baskett, F., and Pratt, V.
The SUN Hardware Architecture.
Technical Report 229, Computer Systems Laboratory, Stanford University, March, 1982.

[Bobrow 81] Bobrow, D. and Stefik, M.
The LOOPS Manual.
Technical Report KB-VLSI-81-13, Knowledge-Based VLSI Design Group Memo, 1981.

[Brown 82] Brown, H. and Stefik, M.
Palladio: An Expert Assistant for Integrated Circuit Design.
Technical Report HPP-82-5, Stanford University Heuristic Programming Project, 1982.

[Bryant 81] Bryant, R.
A Switch-Level Model of MOS Logic Circuits.
In *Proceedings of the First International Conference on VLSI.* Academic Press, 1981.

[Clark 82] Clark, J.
The Geometry Engine: A VLSI Geometry System for Graphics.
In *Proc. SIGGRAPH '82.* ACM, 1982.

[Davis, T. and Clark, J. 82]
Davis, T. and Clark, J.
YALE User's Guide: A SILT-Based VLSI Layout Editor.
Technical Report 223, Computer Systems Laboratory, Stanford University, October, 1982.

[EL Gamal 82] El Gamal, A.
Notes on Gate Array Wiring Area Prediction.
1982.
ISL VLSI File #102282.

[Floyd 82] Floyd, R. W. and Ullman, J. D.
The Compilation of Regular Expressions into Integrated Circuits.
JACM 29(3):603-622, July, 1982.

[Genesereth 80] Genesereth, M., Griner, R. and Smith, D.
MRS Manual.
Technical Report HPP-80-24, Stanford University Heuristic Programming Project, 1980.

[Gerring 81] Gerring, P.
HILGA: A high level, representation independent, object-oriented graphics package for Interlisp-D.
Technical Report KB-VLSI-81-12, Knowledge-Based VLSI Design Group Memo, 1981.

[GreeneEl Gamal 82]
Greene, J. and El Gamal, A.
Area and Time Penalties for Restructurable VLSI Arrays.
1982.
submitted to the Third Caltech Conference on VLSI, March 1983.

[Gross 82] Gross, T.R. and Hennessy, J.L.
Optmizing Delayed Branches.
In *Proceedings of Micro-15.* IEEE, October, 1982.

[Hemachandra 82]
Hemachandra, L.
GRY: a PLA Minimizer.
1982.
Unpublished memorandum, Stanford Univ., Dept. of C. S.

[Hennessy 83] Hennessy, J.L. and Gross, T.R.
Postpass Optimization of Code in the Presence of Pipeline Constraints.
ACM Trans. Prog. Lang. and Sys., 1983.
accepted for publication.

[HennessyJouppiPrzybylski 82]
Hennessy, J.,Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J.
MIPS: A Microprocessor Architecture.
In *Proceedings of Micro-15*, pages 17-22. IEEE, October, 1982.

[Karplus 82] Karplus, K. J.
CHISEL: an Extension to the Programming Language C for VLSI Layout.
PhD thesis, Stanford Univ., Sept., 1982.

[Novak 82] Novak, G.
GLISP: A High-Level Language for A. I. Programming.
In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, 1982.

[Smith 82] Smith, L., Saxe, T., Newkirk, J., Mathews, R.
A New Area Router, the LRS Algorithm.
In *Proc. ICCC*, pages 256-259. New York, September, 1982.

[Stefik 82a] Stefik, M., Bobrow, D., Bell, A., Brown, H., Conway, L. and Tong, C.
The partitioning of concerns in digital system design.
In Penfield, P. (editor), *Proceedings, Conference on Advanced Research in VLSI.* Artech House, 1982.

[Stefik 82b] Stefik, M., Bell, A. and Bobrow, D.
Rule-Oriented Programming in LOOPS.
Technical Report KB-VLSI-82-19, Knowledge-Based VLSI Design Group Memo, 1982.

[Tong 82] Tong, C.
A Framework for Design.
Technical Report HPP-82-11, Stanford University Heuristic Programming Project, 1982.

[Trickey 82] Trickey, H. W.
Good Layouts for Pattern Recognizers.
IEEE Trans. on Computers C-31(6):514-520, June, 1982.

[TrickeyUllman 82]
Trickey, H. W. and Ullman, J. D.
A Regular Expression Compiler.
In *Proc. COMPCON 82.* IEEE, Feb., 1982.

[Ullman 82a] Ullman, J. D.
Combining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits.
Technical Report STAN-CS-82-927, Dept. of C. S., Stanford Univ., Sept., 1982.

[Ullman 82b] Ullman, J. D.
Algorithms and Complexity Issues in VLSI.
1982.
Department of Computer Science, Stanford Univ.

[Wolf 82] Wolf, W., Newkirk, J., Mathews, R., Dutton, R.
Dumbo, a Schematics-to-Layout Compiler.
1982.
submitted to the Third CalTech Conference on VLSI, Mar. 1983.

Publications

[Brown 82] Brown, H. and Stefik, M.
Palladio: An Expert Assistant for Integrated Circuit Design.
Technical Report HPP-82-5, Stanford University Heuristic Programming Project, 1982.

[Davis, T. and Clark, J. 82]
Davis, T. and Clark, J.
YALE User's Guide: A SILT-Based VLSI Layout Editor.
Technical Report 223, Computer Systems Laboratory, Stanford University, October, 1982.

[DavisClark 82] Davis, T. and Clark, J.
SILT: A VLSI Design Language.
Technical Report 226, Computer Systems Laboratory, Stanford University, October, 1982.

[EL Gamal 82] El Gamal, A.
Notes on Gate Array Wiring Area Prediction.
1982.
ISL VLSI File #102282.

[Floyd 82] Floyd, R. W. and Ullman, J. D.
The Compilation of Regular Expressions into Integrated Circuits.
JACM 29(3):603-622, July, 1982.

[GreeneEl Gamal 82]
Greene, J. and El Gamal, A.
Area and Time Penalties for Restructurable VLSI Arrays.
1982.
submitted to the Third Caltech Conference on VLSI, March 1983.

[Gross 82] Gross, T.R. and Hennessy, J.L.
Optmizing Delayed Branches.
In *Proceedings of Micro-15.* IEEE, October, 1982.

[Hemachandra 82]
Hemachandra, L.
GRY: a PLA Minimizer.
1982.
Unpublished memorandum, Stanford Univ., Dept. of C. S.

[Hennessy 83] Hennessy, J.L. and Gross, T.R.
Postpass Optimization of Code in the Presence of Pipeline Constraints.
ACM Trans. Prog. Lang. and Sys., 1983.
accepted for publication.

[HennessyJouppiPrzybylski 82]
Hennessy, J.,Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J.
MIPS: A Microprocessor Architecture.
In *Proceedings of Micro-15*, pages 17-22. IEEE, October, 1982.

[Karplus 82] Karplus, K. J.
CHISEL: an Extension to the Programming Language C for VLSI Layout.
PhD thesis, Stanford Univ., Sept., 1982.

[MathewsWatson 82]
Mathews, R. Watson, I. and Wolf, W.
Testing Chips using ICTEST Version 2.5.
1982.
ISL VLSI File # 012782.

[Noice 82] Noice, D., Mathews, R., Newkirk, J.
A Clocking Discipline for Two-Phased Digital Systems.
In *Proc of ICCC*, pages 108-111. New York, September, 1982.

[Saxe 82] Saxe, T.
Testing the High Yield Memory.
1982.
ISL VLSI File #821019.

[Smith 82] Smith, L., Saxe, T., Newkirk, J., Mathews, R.
A New Area Router, the LRS Algorithm.
In *Proc. ICCC*, pages 256-259. New York, September, 1982.

[Tong 82] Tong, C.
A Framework for Design.
Technical Report HPP-82-11, Stanford University Heuristic Programming Project, 1982.

[Trickey 82] Trickey, H. W.
Good Layouts for Pattern Recognizers.
IEEE Trans. on Computers C-31(6):514-520, June, 1982.

[TrickeyUllman 82]
Trickey, H. W. and Ullman, J. D.
A Regular Expression Compiler.
In *Proc. COMPCON 82.* IEEE, Feb., 1982.

[Ullman 82a] Ullman, J. D.
Combining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits.
Technical Report STAN-CS-82-927, Dept. of C. S., Stanford Univ., Sept., 1982.

[Ullman 82b] Ullman, J. D.
Algorithms and Complexity Issues in VLSI.
1982.
Department of Computer Science, Stanford Univ.

[Watson 82] Watson, I., Newkirk, J., Mathews, R. and Boyle, D.
ICTEST: A Unified System for Functional Testing and Simulation of Digital ICs.
In *Proc. Cherry Hill Int'l Test Conference.* Philadelphia, 1982.

[Wolf 82] Wolf, W., Newkirk, J., Mathews, R., Dutton, R.
Dumbo, a Schematics-to-Layout Compiler.
1982.
submitted to the Third CalTech Conference on VLSI, Mar. 1983.

Heuristic Programming Project
Report No. HPP-82-5

Palladio: An Expert Assistant for Integrated Circuit Design

HAROLD BROWN
HEURISTIC PROGRAMMING PROJECT
DEPARTMENT OF COMPUTER SCIENCE
STANFORD UNIVERSITY

MARK STEFIK
VLSI SYSTEM DESIGN AREA
XEROX PALO ALTO RESEARCH CENTER

The Stanford University component of this research is funded by the
Defense Advanced Research Projects Agency under Contract MDA-903-80-c-007.

Palladio: An Expert Assistant for Integrated Circuit Design[1]

Harold Brown
Department of Computer Science, Stanford University

Mark Stefik
VLSI System Design Area, Xerox PARC

Abstract. We are currently developing a sys.em, Palladio, which serves as a vehicle for experimentation with various integrated circuit design methodologies and with knowledge-based expert system design aids. This paper describes the basic design concepts underlying Palladio, the overall architecture of Palladio and the current development status.

1. Introduction

The Palladio[2] system is a framework for experimentation with integrated circuit (IC) design methodologies, expert system techniques, and symbolic circuit simulation concepts. Palladio serves as the focus for the Knowledge-based VLSI Project (KB-VLSI project), a collaborative activity between the Heuristic Programming Project, Stanford University and the VLSI System Design Area, Xerox Palo Alto Research Center.

The KB-VLSI project is concerned with understanding the processes by which artifacts, in particular, integrated circuits, are designed. The long-term goals of the project are:

> Identify and articulate the expert knowledge used in integrated circuit design. An objective here is to gain an understanding of the design process and to develop cognitive models of the process.
>
> Develop methods for representing and reasoning with design knowledge. Such reasoning involves design constraints, goals, and tradeoffs.
>
> Develop knowlege-based expert systems for assisting in the IC design, test and debug cycle. The systems include aids for entering and recording IC design specifications and aids for transforming abstract design specifications into more detailed specifications.

[1] The Stanford University component of this research is funded by the Defense Advanced Research Projects Agency under Contract MDA-903-80-c-007.

[2] *Andrea Palladio* (1518-1580) was an Italian Renaissance architect of great reknown. He is perhaps best known because he developed a methodology of proportion and formal architectural style that has become known as *classical* architecture. In a sense, he was the first *knowledge engineer* of design principles and his influential published works are still in print four hundred years after his death.

Palladio is the primary research vehicle for the KB-VLSI project.

2. Palladio's Model of the Design Process

An IC design process can be viewed as the creation of behavioral and structural specifications of a circuit. This usually involves a sequence of transformations from abstract specifications of the behavior and structure of the circuit to more detailed implementation specifications. For example, the design of a combinational logic circuit may involve first transforming a specification of the circuit in terms of boolean equations which relate the inputs and outputs into a specification in terms of logic gates and interconnection networks, and then transforming this latter specification into a layout specification expressed in terms of "colored" rectangles.

A useful metaphor for this transformation process is that design is search [7]. The designer searches in a solution space of implementation specifications. Moves in this space are design decisions. Each design decision involves considering alternative implementations, testing the alternatives against the constraints and goals imposed by the abstract specifications, and using tradeoffs to differentiate between "satisficing" alternatives and to resolve conflicts between incompatible constraints and goals. The design decision process is difficult because: (a) the solution space is large, (b) the generation of alternative solutions is expensive, (c) only partial information is available, (d) it is not possible to predict all of the consequences of a decision.

2.1. Design Hierarchies

IC designers have, in part, coped with the difficulty of making design decisions by exploiting hierarchies in the design process. One powerful hierarchical technique is to decompose a device into semi-independent subdevices and to focus attention on each subdevice individually. For example, a 4-bit register can be considered as four 1-bit registers and their interconnections. The focus on a subdevice reduces the size of the solution space under consideration.

The device-subdevice hierarchy is only one way of partitioning the design process. Design using *description levels* (abstract models of circuits) is a complementary way to do it. Each description level provides languages for describing the behavior and structure of a device which suppress particular details of physical implementations of the device. The use of description levels reduces the complexity of the elements in a solution space and makes the generation and comparision of alternatives less expensive.

Description levels also permit a designer to partition concerns by concentrating on subclasses of design decisions. For example, at an architectural level a designer can work out certain storage and communication decisions before worrying about power considerations. The derivation of useful design description levels requires significant domain-specific knowledge -- a sort of "engineering of knowledge" [8].

We are currently experimenting with four description levels in the Palladio system: *Layout*, *Clocked Primitive Switches* (CPS), *Clocked Registers and Logic* (CRL) and *Linked Module Abstraction* (LMA). Collectively, these levels factor the concerns of a digital designer [6].

The most widely used description level in integrated circuit design is the artwork or layout level. This level describes integrated circuits in terms of "colored rectangles" (representing material on a chip) that can be composed to build up large designs. Associated with the colored rectangle terms of the layout level is a set of *composition rules*, called layout design rules. The layout composition rules provide a simple *shallow model* of composition that is based on a *deep model* of electrical properties and fabrication tolerances. If designers follow these rules, their designs are guaranteed to have adequate physical spacing on a chip [3, 4].

The layout description level has several important properties which make it useful for the synthesis of designs. First, primitive terms can be combined to form larger terms and subsystems. Second, there are rules of composition that define the allowed compositions of these terms. These rules apply both to composite objects and primitive terms. Third, there is a well characterized set of *bugs* that are avoided when the composition rules are obeyed. At the layout level, these bugs correspond to the function and performance problems caused by incorrect physical spacing.

All of our proposed more abstract description levels have properties analogous to those of the layout level. The CPS level distinguishes between different uses for logic and is concerned with the digital behavior of a system. Different uses of logic include steering logic, clocking logic, and restoring logic. The composition rules at this level prevent bugs of non-digital behavior caused by charge sharing and invalid switching levels. The CRL level is concerned with the composition of combinational and register logic. The composition rules at the CRL level preclude various bugs related to clocking in a two-phase system. The LMA level is concerned with the sequencing of computational events in a digital system. It describes the paths along which data can flow, the sequential and parallel activation of computations, and the distribution of registers. The composition rules at the LMA level preclude bugs such as starting computations before the data are ready, and deadlock bugs that arise from the improper use of shared modules.

2.2. Design Knowledge Bases

Much of the design of ICs is done by using parts of existing designs, possibly with modification. This technique exploits the fact that there are common constructs used in many circuits; for example, registers, NAND gates and input-output pads. The use of previously designed (and debugged) components in a current design is analogous to the use of subroutine packages in software development.

In Palladio, knowledge about previously defined circuits is kept in community knowledge bases. These knowledge bases can contain not only exisiting designs but also collections of knowledge about the composition and the optimization of circuit components. For example, at the CPS level we are developing a knowledge base which includes a collection of prototype logic gates, a set of rules that define the allowed composition of these gates and a set of optimization rules for reducing various costs of circuits composed of networks of gates.

The use of community knowledge bases in Palladio is supported by the LOOPS system [1]. LOOPS is an object and data oriented programming system implemented in the Interlisp [9] programming environment. LOOPS was created, in particular, to support a design environment in which there are community knowledge bases that people share, and to which they can add incremental updates.

2.3. Design Evòlution

The design of a complex artifact such as an integrated circuit is an evolutionary process that follows an iterative cycle: create a candidate design — test the candidate design against current requirements — modify the design and/or requirements to create a new candidate design. An IC design system should have facilities for interactive simulation to provide a rapid feedback between proposed changes and their exercise on test cases.

Within the Palladio framework, we have begun experiments with interactive, rule-based symbolic circuit simulators. These simulators use symbolic reasoning on a hierarchy of behavioral and structural specifications for a circuit in order to predict the outputs of the circuit given a set of inputs. The simulators include a dynamic display capability, *i.e.*, "animated simulation cartoons." Our objective is to develop a design environment based on simulators, interactive editors, and debugging tools comparable in power and flexibility (and in concept) to, for example, the Interlisp Programmer's Assistant [10].

3. Palladio's Architecture

A major purpose of the Palladio system is to provide a common starting point and a framework for the research activities of the KB-VLSI project. As such, Palladio must be sufficiently general so that it can be easily extended as our research continues. At the same time, Palladio must admit sufficient specialization so that we can rapidly experiment with particular design concepts. To achieve these goals we have used a knowledge based architecture for Palladio.

The overall architecture of the Palladio system is shown in Figure 1.

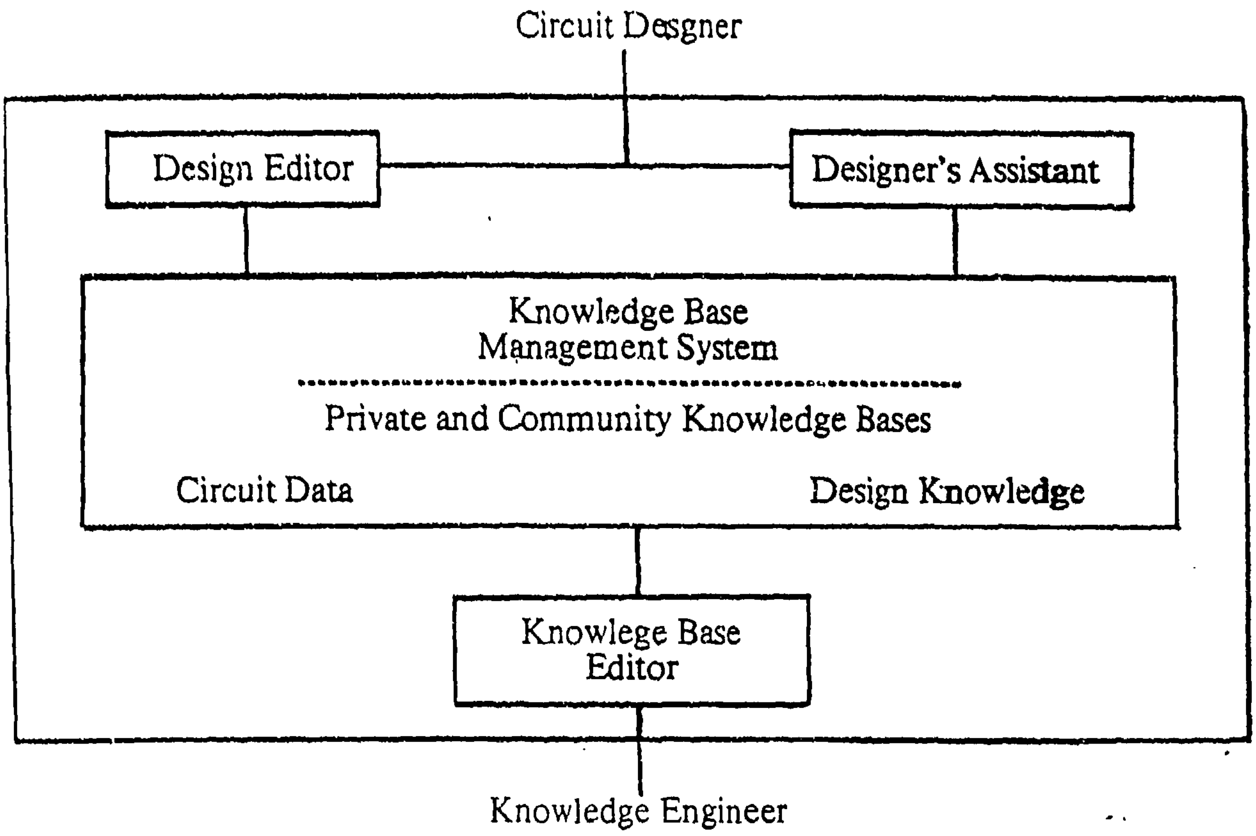


Figure 1. Palladio System Block Diagram

There are two classes of users of Palladio: knowledge engineers and circuit designers. Knowledge engineers use the *knowledge base editor* to enter concepts and rules of design that define Palladio's design methodologies. This knowledge is kept in community knowledge bases. Circuit designers interact with the *design editor* and *designer's assistant* to create circuit designs. The design editor enables a designer to enter and modify circuit descriptions at various levels of description. The editor uses the composition rules of each design methodology to assure that the design is "legal" with respect to that methodology.

The designer's assistant is an active element that can propose design decisions. The two programs are integrated with a single graphics interface from which the user can control the activity and participation of the designer's assistant. The design editor, designer's assistant, and knowledge base editor all communicate with the knowledge base via the *knowledge base management* component of LOOPS.

4. Current Status

Most of the supporting framework for the Palladio system is currently in place. The LOOPS programming system [1], has been fully implemented. A high-level, object-oriented graphics package has been developed for the Xerox Dolphin personal computer, the development machine for the KB-VLSI project. This package, HILGA [2], is interfaced with the LOOPS system. The GLISP language [5], has been interfaced with LOOPS. GLISP provides LOOPS with optimized data and procedure access.

Prototype community knowledge bases for the CPS and LMA description levels are substantially completed. The initial knowledge bases for the layout and CRL levels are under development. A rule-based design editor for the CPS level is partially implemented.

A prototype "animated" simulator for the LMA level has been implemented. The implementation of an interactive simulator for the CPS level has been started.

Research has been initiated on expert system design assistants to aid in transforming abstract design specifications into more detailed specifications [11]. This work includes research on the use of tradeoffs in the design process.

By the end of this year we plan to have enough of the Palladio system in place so that it can be used to create designs of simple, "student - level" integrated circuits.

5. Concluding Remarks

An important purpose for Palladio is as a vehicle for community building. Opportunities for developing systematic bodies of design knowledge will appear in many parts of the VLSI design community. Although knowledge engineering provides effective techniques for capturing and debugging this knowledge, these techniques are not widely understood or practiced in the VLSI community. In particular, there is a shortage of trained knowledge engineers and suitable computers for this work. Because of the intellectual and computational hurdles, it is unlikely that expert systems will be widely available in the community for several years. It is our intention to keep this part of our research open and to invite experimentation with our facilities and participation by other members of the VLSI design community as opportunities arise.

6. Acknowledgements

The development of the Palladio system is a team effort involving all of the members of the KB-VLSI project. The current members of the project are: Gordon Foyster, Phillip Gerring, Gordan Novak, Narinder Singh, Christopher Tong and the first author at Stanford University; and Alan Bell, Daniel Bobrow, Lynn Conway and the second author at Xerox Palo Alto Research Center.

References

1. Bobrow, D., Stefik, M. A virtual machine for experiments in knowledge representation. submitted to *AAAI Conference*, 1982.

2. Gerring, P. HILGA: A high level, representation independent, object-oriented graphics package for Interlisp-D. Knowledge-Based VLSI Design Group Memo KB-VLSI-81-12 (working paper).

3. Lyon, R. Simplified design rules for VLSI layouts. *Lambda*, First quarter, 1981.

4. Mead, C., and Conway, L. *Introduction to VLSI Systems*. Addison-Wesley Publishing Company, 1980.

5. Novak, G. GLISP: A high-level language for AI programming. submitted to *AAAI Conference*, 1982.

6. Stefik, M., Bobrow, D., Bell, A., Brown, H., Conway, L., Tong, C. The partitioning of concerns in digital syste n design. in Paul Penfield (Ed.), *Proceedings, Conference on Advanced Research in VLSI*, Artech House, January 1982.

7. Simon, H. *The Sciences of the Artificial.* MIT Press, 1969.

8. Stefik, M., Conway, L. The engineering of knowlege for an expert system: A case study. (in preparation).

9. Teitelman, W. *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

10. Teitelman, W. Automated programmering – The programmer's assistant. *Proceedings of the Fall Joint Computer Conference*, December 1972.

11. Tong, C. A framework for design. submitted to *AAAI Conference*, 1982.

SILT:

A VLSI Design Language

Tom Davis and Jim Clark

Technical Report No. 226

October 1982

This research was supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680.

SILT:

A VLSI Design Language

Tom Davis and Jim Clark

Technical Report No. 226

October 1982

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

SILT is an efficient, medium-level language to describe VLSI layout. Layout features are described in terms of a coordinate system based on the concept of relative geometry. SILT provides hierarchical cell description, a library format for parameterized cells with defaults for the parameters, constraint checking (but not enforcement), and some name control. It is designed to be used with a graphical interface, but can be used by itself.

Key Words and Phrases: VLSI, design tools, layout editors, relative geometry

Table of Contents

List of Figures

1. Introduction

1.1 SILT Overview

In many endeavors, especially VLSI design, what seems to be the last 10% of the work often takes 90% of the time. It is often not too difficult to lay out the initial circuit, but "small" modifications can take an enormous amount of time. One of the main goals of the SILT VLSI layout language is to make such modifications easier. Another goal is to provide a convenient form for general library cells that provides for some "stretch" in the cells. Finally, SILT's naming conventions are designed help the user keep track of the names involved in a large hierarchical circuit.

It is easiest to describe SILT as a language by analogy with programming languages. CIF corresponds to machine language, CLL (a Stanford language that is essentially CIF with symbolic names instead of numbers, see [5]) corresponds to an absolute assembler, and SILT corresponds to a relocatable assembler. SILT is not a "silicon compiler" in that it is descriptive rather than procedural. Because of this, it is not too hard for the user to figure out exactly what geometry will be produced by a given set of instructions. Geometry produced by SILT would bear roughly the same relation to the geometry produced by a true silicon compiler that machine code produced by a language assembler would to that produced by a full-blown compiler.

SILT has some features not present in CIF or CLL. The most important is probably the parameterization of symbols which can be extremely important for a library format. SILT's local names and its method of exporting only certain names outside symbols help to control the size of the name space. Finally, SILT provides some mechanism for constraint checking that is done automatically when symbols are expanded.

SILT's syntax looks much like that of a block structured language such as ALGOL or PASCAL. A SILT file is a series of symbol definitions followed by calls on those symbols. A symbol includes a parameter list, some declarations (including, perhaps, definitions of other symbols), and a series of symbol calls. The symbol calls can be on previously defined symbols or on primitive symbols, such as rectangles, contact cuts, and butting contacts. The scoping rules for names are similar to those in PASCAL, so symbols and variables declared within another symbol are local to it. The mechanism for passing data to and from a symbol will be discussed in greater detail later. There are advantages and disadvantages to this, which are described in 3.1.

SILT can be used by itself to lay out circuits, but it is designed to be used with a graphical front end.

The language is designed to be an interchange form, however, so it in not tied to any particular graphics editor.

It is possible to use SILT using input generated by other graphical editors. The only requirement is that they be able to produce CIF output. In SILT, certain symbols can be declared to be external. A symbol that is so declared can be used in much the same way as other SILT symbols. There are some restrictions, however.

SILT is not a complicated language -- in most cases, the designer should be able to figure out exactly what geometry will be generated by any particular fragment of code. It has no powerful built-in primitives such as "insert 16-bit ALU" or "route signal1<0:15> to signal2<16:31>". The "16-bit ALU" may exist in a library, but the user will have to do the routing for the second example. There are only a few features found in SILT that are not found in some other language -- SILT primarily makes a convenient set of features available within a single language.

1.2 The Implementation

At present, two programs exist for the conversion of SILT to CIF and back again. Both are written in PASCAL. The CIF to SILT converter should be quite portable, but the SILT to CIF program is based on Hennessy's parser generator (see [2]), so porting the SILT program would also require porting the parser generator as well.

The code is written to run under both TOPS-20 and UNIX. Some minor changes must be made in the code to transfer it from one operating system to another. The source code is the TOPS-20 version, but there are instructions for the edits that must be performed to make the UNIX version at the beginning of the file.

1.3 Using this Document

The easiest way to learn SILT (or any other programming language, for that matter) is by looking at examples. A series of examples is provided in appendix I. The exact syntax for SILT can be found in appendix II, which includes pointers back into the document for discussions of the associated semantics. Finally, there is a list of the SILT reserved words in appendix III.

Page 4 blank
Page 5 not available

1.4 Using SILT

SILT currently runs under TOPS-20 (Hedrick's PASCAL) and BERKELEY UNIX (BERKELEY PASCAL). The procedure for using it is similar in both cases. In the TOPS-20 version, when the program starts up, the user will be asked for the name of an input file and an output file. The input file is the SILT source, and the output file will contain the CIF generated. No default extensions are assumed, so the whole file name must be typed in both cases. Any errors encountered are printed out on the terminal. The general philosophy followed by SILT is that it attempts to recover from as many errors as possible. Thus, when the input file contains errors, SILT is not guaranteed to work and run-time errors occasionally occur. The hope is that enough error diagnostics are generated so that the user can correct the errors and try again.

For the UNIX implementation, the standard input and output are used, and most of the errors are recorded in a special file called "errors". A few messages are sent to the terminal. As in the TOPS-20 version, the entire file names must be specified. The standard extensions that most people use are ".slt" for SILT files, and ".cif" for CIF files.

When the UNIX version is used on the VAX, SILT files can be used with the "C" preprocessor. The SILT assembler does not itself call on the preprocessor, but it ignores any lines beginning with the character " # ". The main use of this is for the inclusion of files.

A graphical front-end for this language can present the symbol on the screen together with the relative points. In addition to the usual commands for adding, deleting, and moving rectangles, the user can add, delete, and move the relative points. When a relative point is moved, all associated geometry is adjusted. The graphical editor must also include commands to associate edges of rectangles with relative points.

In fact, two graphical SILT-based editors have been written. The first was purely experimental, called ALE (see [4]), and was used to experiment with various techniques to interract graphically with SILT constructs. Later, based upon the ALE experiences, another editor, called YALE (see [1]) has been implemented on the SUN workstations. YALE does not implement all of the features of SILT, but does implement the more important ones. All of YALE's input and output is done in a subset of SILT.

3. Names and Data Types

3.1 Names and Scoping

All user-defined names in SILT begin with a letter and are followed by any number of letters, digits, or the underscore character ("_" = ASCII 137B). No case distinction is made, so "AbC5" and "abC5" represent the same name. Users may not use any of the SILT reserved words, which are listed in appendix III.

SILT's scoping rules are similar to those in PASCAL, with one important exception. If SILT's symbols are thought of as PASCAL procedures or functions, then the variables visible with a given nesting of symbols would be the same ones visible within the same nesting of PASCAL procedures. The exception is SILT's export mechanism. This allows a symbol to make a certain set of internally defined symbols visible outside it. This topic is fully discussed in section 3.5.

Future versions of SILT may not allow such complete freedom in nesting functions and in the scope of names. For efficiency, every time SILT expands a symbol, it keeps track of the parameters passed to the symbol and to the CIF code produced. Every time the symbol is expanded, a check is made to see if it has been expanded before with the same parameters. If so, no expansion is done, and a pointer to the already expanded symbol is returned. If a symbol depends on variables not in the parameter list, errors will occur. Future versions of SILT may not allow nesting of functions, and will thus have only two kinds of variables -- local and global. Thus, it is not recommended that full advantage be taken of SILT's nesting mechanism.

3.2 Variable Types

SILT deals with four basic kinds of variables: x- and y-relative points (described in the previous chapter), scalar values and signals. These are declared as xvar, yvar, scalar, and signal, respectively. Variables of type xvar, yvar and signal are simply stored as a single real number, but signals are more complicated. A signal is a collection of triplets, where each triplet consists of an x- and y- coordinate together with an optional process layer (metal, diffusion, polysilicon, etc.).

Scalars are meant as a catch-all to include such things as iteration variables, pull-up ratios, and power requirements. Scalars are not altered when a symbol is transformed (x-relative points are changed to y-relative points when the symbol is rotated 90 degrees).

Signals are meant to be used to identify particular points of the geometry so that various constraints can be checked. A user may, for example, insist that point "a" of signal "b" be connected to point "c" of signal "d" (see 5.12). Signals are transformed as a pair of relative points when their symbol is transformed. Vectors of signals such as "addr<0:23>" can be declared, and one can refer to the components of particular signals with expressions like "a_in.x" or "addr<0>.y". The reason that a signal consists of possibly more than one triplet of values is that the same electrical signal is often available at different points within the symbol. When one asks to have one signal connected to another, it does not matter to which of these points a connection is made. This is especially useful for symbols that have a bus passing through them, and hence each signal on the bus will be available at both ends of the symbol.

All the variables described above can be combined in the usual way with arithmetic operations and constants. SILT makes no checks to ensure that the expressions formed make sense -- it will cheerfully allow the user to multiply relative points or to subtract an x-relative point from a y-relative point.

All numbers in SILT are stored internally as real values, and thus division does not round. SILT also has the integer functions "div" and "mod". "x mod y" is evaluated as "float(round(x) mod round(y))", and "div" is handled similarly. Following are a few examples of valid SILT arithmetic expressions:

```
abc + (power*width)
(q mod r) - xyz / (7.3*((p+q) div r))
```

In addition to "+", "-", "*", "/", "div", and "mod", SILT includes an (experimental) function "bitop" (standing for "bit operation"). It is a function of two integers (which are produced by rounding as for "div" and "mod", above), and allows one to determine whether a given bit in the binary expansion of a number is "1" or "0". "bitop(number, bitposition)" yields the numeric value 1 or 0. The least significant bit is number zero, so "bitop(5,0)" yields 1, "bitop(5,1)" is 0, and so on.

In a sense, SILT also deals with boolean values, although there is no way to save such a value in a variable. Arbitrarily complicated boolean expressions like "a >= b AND ((NOT b>=7) OR (c+e <= d))" can be formed and used in both conditional statements (see section 5.8) or constraints (see section 4.5). At present, the comparitors ">" and "<" are not available because of some restrictions of the parser, but they can be implemented using "NOT" with ">= " and "<= ".

3.3 Symbol Names

All the geometry in a circuit is defined in terms of symbols, the bodies of which are made up of primitive calls and calls on other symbols. Symbols may be defined within other symbols, and symbols so defined are "local" to their containing symbol. Locals within different symbols may have the same names.

The entire SILT file can be thought of as a distinguished symbol that is different from other symbols in that it has no parameters and that it is automatically expanded once at (0, 0). The "file" in SILT is to its symbols much as the "program" in PASCAL is to its procedures.

3.4 Instance Names

A single symbol can be instantiated many times, and any number of particular instantiations may be named It is common to lay out one- or two-dimensional arrays of symbols, so SILT allows array-like instance names. A symbol call is named if it is preceded by an instance name followed by "::". For example, if "foo" has been declared as a symbol, then it might be instantiated in the following ways:

```
place foo() at ...               (* no instance name here *)

a:: place foo() at ...

b[2]:: place foo() at ...

for i := 0 to 7 do
  for j := 0 to 7 do
    c[i,j]:: place foo() at ...
```

In arrays of instances, the indices are rounded to the nearest whole number(s). No instance names (including arrays of instances) need to be declared ahead of time. An array of instances must all correspond to the same symbol, although the symbol in question may be called with different parameters in each instance.

3.5 Exports

All variables defined within a symbol are local to the symbol, unless they are specifically exported using an export declaration. Information exported from a symbol is visible only within the symbol that called it. Thus only information that is in some sense important is visible outside a symbol. If symbol "a" calls symbol "b" and symbol "b" calls symbol "c", then the exports of symbol "c" are not automatically visible in the body of symbol "a" unless "b" also specifically exports them. The example in Figure 3-0 below illustrates the code required to make "c"'s export visible inside symbol "a" (as b_inst.c_export):

The types of things that are exported often include some subset of the relative points, certain connection points (signals), power requirements, and information about the size of the cell. The size of the cell is automatically exported even if the user does not specifically ask for it ("xmin", "xmax", "ymin" and "ymax" are the values automatically exported). The example that follows illustrates the method for accessing exported names. If a symbol is placed without an instance name, its exports are not accessible.

Figure 3-1: Exporting values

```
symbol c();
scalar c_export;
export c_export;
...
symbol b()
scalar c_export;
export c_export;
    begin
    c_inst:: place c() at (0, 0);
    c_export := c_inst.c_export;
    ...
    end;
symbol a()
    begin
    b_inst:: place b() at (0, 0);
    ...
    end;
...
```

3.6 CIF Names

SILT also has a provision for passing certain names to the CIF file produced. These names can be attached to a point and optionally to a layer. See section 5.9.

4. Symbol Declaration

Every SILT symbol (including the entire SILT file itself) has a header, a series (possibly empty) of declarations, and then a list of symbol commands. This chapter discusses the header and declarations of a symbol definition.

SILT input is free-form in the spirit of PASCAL. Spaces, tabs, and carriage-returns can appear anywhere except within an identifier or a reserved word.

Comments can be imbedded anywhere in a SILT file where a space could appear, and are made up of arbitrary text, surrounded by "(*" and "*)". Comments may be nested, making it possible to comment out arbitrary chunks of SILT code.

All symbols must be defined before they are used -- there is nothing that would correspond to a PASCAL "forward" declaration. Instance names need not be declared ahead of time, but all other variables must be.

4.1 Symbol Parameters

It is possible to pass any number of relative point values to a symbol via the parameter list. SILT also allows the user to pass other values to a symbol, such as scalars representing power requirements or pullup ratios. Each parameter should have a default value associated with it that is usually chosen to produce a cell of minimum size. See section 4.4. Since the minimum cell is usually what is desired, parameter specification will be the exception rather than the rule. For this reason, both key calls and positional calls are implemented in SILT. If the symbol foo has a header that looks like:

```
     symbol foo(xvar x0, x1, x2);
```

Then the following two forms are valid calls on the symbol:

```
     place foo(x0=3, x2=15) at (0, 0);
     place foo(3, 4, 18) at (0, 0);
```

In the first case, x1 will get the default value, whatever it is. In the second, x0, x1, and x2 would be assigned to 3, 4, and 18, respectively.

4.2 Declaration of Variables

The symbol header is followed by an optional constant declaration and then a series of variable declarations. Constants serve the same purpose as constants in PASCAL, and their values cannot change after their initial declaration. A constant declaration might look something like:

```
constant a := 1; b := 6; c3po := 11.38;
```

Any variables that are to be local to the symbol are declared next as in the following example:

```
xvar x0, x1, x2;
yvar y0;
scalar power, indx;
signal a, addr<0:23>, vdd, gnd;
```

The variable declarations may appear in any order.

4.3 Exports

Any variable declared within a symbol can be exported by including its name in an export declaration. In addition, variables from the parameter list can also be exported. (This last is not so silly as it seems -- since various defaults may be taken, the user would otherwise have no easy way to find out some of the relative point positions.) As was stated before, every symbol effectively includes the following:

```
xvar xmin, xmax;
yvar ymin, ymax;
export xmin, xmax, ymin, ymax;
```

The general export declaration looks exactly like the example above. If a signal vector is exported, only its name is necessary -- the size has already been declared. The following example illustrates all the possibilities:

```
symbol foo(yvar y9);
xvar a, b;
signal c, d<0:5>, e;
export a, c, d, b, y9;
```

4.4 Defaults

A default declaration is made up of a list of assignment statements that are interpreted in a special way. The left-hand-side (LHS) of an assignment must be a member of the formal parameter list, but the right-hand-side (RHS) can be an arbitrary expression. The only constraint is that all variables in the RHS must be known by the time the expression is evaluated. A typical set of default values (in this case for Figure 3-0 above) will look something like this:

```
default x0 := 9; x1 := x0 + 6; x2 := x0 + 9; x3 := x1 + 17;
```

Page 14 blank

The form above makes the relative relations among the points clear -- x0 is relative to the origin, x1 is relative to x0, and so on.

SILT expands a symbol's parameter list by going through the default declarations in order, and if the LHS variable is unknown, then the RHS is evaluated and assigned to it. If the LHS is known, SILT advances to the next assignment in the default list. If the symbol having the default declaration in the last paragraph were called with x0 = 8 and x2 = 11, then the symbol would be expanded with x0 = 8, x1 = 14, x2 = 11, and x3 = 31.

4.5 Constraint Declarations

It is possible to declare within any symbol a set of constraints to be checked when the cell is instantiated. A constraint is entered as two arbitrary expressions separated by "<=" (less-than-or-equal), ">=", "<>" (not-equal) or "=". Various information about design rules, power requirements, or anything else can be included. SILT makes no attempt to force the constraints to be satisfied -- it simply warns the user of a possible error if a constraint violation occurs.

All the boolean expressions in the constraint list are evaluated after the default list has been processed. If errors are discovered, the user is warned, but the SILT assembly continues. SILT makes no attempt to stretch or shrink geometry to force the constraints to be satisfied -- it merely prints a warning if there is an error.

4.6 Symbol Definitions

After any combination of the above declarations, any number of local symbols may be declared. The program fragment below gives the form of a complete symbol definition, where the "..." is replaced by any number of symbol commands which will be discussed in chapter 5.

```
symbol foo(xvar x0, yvar y0, scalar bar);
constant const1 := 7:
xvar x_loc;
signal sig<0:7>;
export x0, x_loc, sig;
default x0 := 5; y0 := 7;
        scalar := x0 - y0 + 2; (* legal, but senseless *)
    begin
    ...
    end;
```

5. The Symbol Body

A variety of commands are available within the body of a SILT symbol. Each is discussed in detail in the following sections.

5.1 Primitive Calls

At present, there are three types of primitive calls available in SILT. These include contact cuts, butting contacts, and boxes (rectangles).. The calling sequence for each of the above types is similar to the calling sequence for general symbols, but there are a few restrictions. In most of the examples in this document, the word "place" has been put in front of each call on either primitive or user-defined symbols. It is optional, and simply serves to make the SILT text easier to read.

Without too much difficulty, it is possible to add other primitive symbol types to the language, should they prove to be important.

5.1.1 Box Calls

The following three examples illustrate possible box calls:

```
place box(1, 4) at (x0+5, y7+wirewidth/2);
place box (x1-x0, 4, poly) at (0, -23);
box (1,4) at (4, 5);  (* "place" is not required *)
```

The first two parameters enclosed in parentheses after "box" are the x-length and y-length. If there is a third parameter, it must be a layer in the set {metal, poly, diff, implant, contact, buried}. It indicates the layer upon which a particular box is to be placed. If no layer is indicated, then the default layer (from the "with" command -- see section 5.7) is used. If the box command appears without a layer specification, and is not within the scope of a with command, then an error occurs.

The last pair of numbers are the x- and y- coordinates of the lower left corner of the box. None of the other transformations (described later) may be applied to a box call.

Box calls cannot be given an instance name.

An alternative form of the box command can be used that substitutes the word "to" for "at". A typical call might look something like this:

```
place box (1, y3+17, poly) to (7, y3+22);
```

In this form, the first pair of numbers serve as coordinates for the lower-left corner and the second two as coordinates for the upper-right corner. This form makes it much easier to see exactly which reference point is associated with each edge, and hence provides a much better graphical interface.

5.1.2 Contact Cuts

Contact cuts come in two flavors -- one connects metal to poly, and the other connects metal to diffusion. A contact cut may be placed as follows:

```
place cut(poly) at (3, 6);
place cut(diff) at (3, 6);
place cut() at (0, 0);
place cut at (0,0);
```

In the third and fourth examples, the layer is chosen in the same way as it is for box calls. If the layer is not in {poly, diff} then an error occurs.

The cut symbol has its origin at the lower-left corner. It can be rotated and flipped, but it is not really necessary for this symbol. See section 5.2.

Contact cuts calls cannot be given an instance name.

5.1.3 Butting Contacts

The butting contact symbol has only one type -- the standard Mead-Conway butting contact (see [3]) oriented horizontally with the diffusion on the left. The origin is at the lower-left corner, and when the cell is placed, any standard transformation can be applied to it. (See the next section for a discussion of transformations.)

Since there is only one type of butting contact, the calls must look like one of the following two examples:

```
place butt() <transformation>;
place butt <transformation>;
```

Like contact cuts and boxes, butting contact calls cannot be given an instance name.

5.2 Transformations

Every symbol that is either user-defined or built-in (butting contact, contact cut) has an origin at the point (0, 0). For the built-in symbols, the origin happens to be at the lower left-hand corner, but this need not be the case for user-defined symbols.

Three different kinds of transformations are allowed: translation, rotation, and reflection. Rotation always takes place about the origin and reflection through either the x- or y- axis. Since all geometry is constrained to be parallel to the coordinate axes, rotations can only occur in multiples of 90 degrees.

Any sequence of translations can be applied to a symbol, applied in the order in which they appear in the SILT description (left to right). The order is important -- a rotation followed by a translation is much different from the same translation followed by the rotation.

Rotations are defined relative to a standard clock face (this idea comes from the CLL language). Imagine the origin of the symbol at the center of a clock face with an arrow super-imposed on it pointing to 12 o'clock. A rotation of 3 leaves the origin in place, but rotates the arrow so that it points to 3 o'clock, and so on. The only rotations allowed are 3, 6, and 9. The syntax for a rotate command is something like "rotated 3".

Reflections are either up-down (up-down means along the y-axis, or across the x-axis) or left-right. The syntax for a reflection must be one of: "flipped ud", "flipped lr" or "flipped rl". The last two are equivalent.

A translation has the form "at (x_trans, y_trans)". This has the effect of translating the origin of the symbol to the point (x_trans, y_trans).

A complete transformation is made up of any number (including zero) of the above, optionally separated by commas. If no transformation is given, the symbol is placed at (0,0) in the standard orientation. Supposing that the symbol "foo" has already been declared, the following are valid symbol calls on "foo":

```
    place foo() at (0, x0+7);
    place foo() flipped ud, rotated 3 at(0,0);
    place foo() at (5, -3) flipped ud;
    foo at (4, 6);
    abc:: foo;
    fooinst:: place foo(power=3) flipped ud at (0, 5) rotated 9;
    place foo() rotated 3;
    place foo() at (3, 4) at (7,5);
```

The next to last example above places the rotated cell at the origin, and the last one is equivalent to:

```
    place foo() at (10, 9);
```

5.3 Symbol Calls

A call on a user-defined symbol must appear after the symbol has been defined. If desired, the call may be preceded by an instance name, followed by "::". Only labelled instances can have their exports referred to later.

The parameter list is optional, and if it is omitted, all the default values will be taken. Parameters may be specified either by a positional call or a key call. If it is a positional call, the parameters are listed in the order in which they appeared in the symbol declaration. If it is a key call, the form is:

```
<formal parameter name> "=" <actual parameter>
```

All calls are by value, not by reference. Following is a short list of examples of symbol calls. Assume that the symbol "foo" has already been declared:

```
    a:: place foo(x1=5, y3=7) at (2, 3);
    b[i+1]:: place foo(x1=x1, power=13) flipped ud at (3, -5);
    q[i,j]:: place foo(x1=i*3) at (x0+q[i-1,j].xmax, 17);
    place foo(in<0:3> = x<0:3>);[1]
    place foo(1, 2, 3) at (3, 4);
```

In the second example in "x1 = x1", the "x1" on the left is the name of the parameter in "foo", and the "x1" on the right is the value of "x1" in the calling symbol. It is impossible to tell (from the example) what variables get set to what values in the final example, since we do not know the order in which the parameters were declared. See also section 4.1.

5.4 The Block Command

The block command is just a convenient way to group together a number of SILT commands for the benefit of a with command, an iteration command, or a conditional command. The syntax is "begin", followed by any number of SILT commands, followed by "end;". SILT blocks can be nested, if desired. An example follows:

```
with poly do
    begin
    place box (1,2) at (3,4);
    place box (3,4,diff) at (5,6);
    place box (7,8) at (9, 10)
    end;
```

[1] A warning is in order here. This is a moderately unusual construction, but there must be a space between the ">" and the "=". Otherwise, the lexical scanner will interpret it as ">=".

5.5 Iteration Commands

SILT contains a simple iteration scheme that behaves almost exactly like a simple form of the PASCAL "for" statement. The following examples give most of the flavor of the command:

```
for i := 1 to 8 do
    for j := 1 to 8 do
        begin
        place box(1,1,metal) at (2*i, 2*j);
        place box(1,1,poly) at (2*i+1, 2*j)
        end;
```

In this first example, "i" and "j" must be declared (presumably as scalars, although they could be of type xvar or yvar).

```
a[0]:: place foo(param=0) at (x0, 0);
for i := 1 to 7 do
    a[i]:: place foo(param=i) at (a[i-1].xmax, 0);
```

The example above places 8 copies of the symbol foo side by side. (In this example, "foo" is assumed to have xmin = 0.) The first instance of "foo" must be placed outside the "for" statement so that the following instances can each refer to the instance to the left. All the instances could have different widths, depending upon what "foo" does with "param".

If the intent is simply to place an array of identical symbols in a linear or rectangular array, use the array command, described in the next section.

5.6 The Array Command

SILT is a rich language, and it would be difficult to implement a graphical front-end that is capable of taking advantage of all SILT's features. One of the more difficult features to implement in its full generality is the iteration command discussed in the last section. Since one of the most common things to do in VLSI design is to lay out an array of symbols, the array commands provide a restricted form of iteration that can fully implemented by a graphical system.

The array command allows the user to place a linear or rectangular array of symbols at a given starting point with a given spacing. All instances of the symbol must be identical. If no spacing is specified, the symbols are placed with x-separation xmax - xmin, and y-separation ymax - ymin. Some examples of the array command follow:

```
place array a[0..7] of foo at (q7, b3+9);
array blat[0..3, 0..5] of foo(6,3-delta,8) at (43, ypos-6);
array bar[0..20] of foo() spacing (5,8) at (11,5);
```

As in other placement commands, the keyword "place" is optional. The array dimensions are described PASCAL-style, and any sort of symbol call can be used after the keyword "of". If "spacing" appears, it is followed by a delta-x, delta-y pair, and the point following the "at" gives the coordinates of the symbol having the smallest x (and y, if there is one) coordinate. It is recommended that this form be used instead of general iteration when it is possible so that the SILT generated can be more easily handled by a graphics system.

The array command:

```
    place array a[0..7] of foo at (0,0);
```

is exactly equivalent to the command:

```
    for i := 0 to 7 do
        a[i]:: place foo at (0,0);
```

except that the variable "i" is not present. One can, however, refer to such things as "a[3].xmin" and "a[5].foo_output" in the usual way.

5.7 The With Command

The "with" command sets the default layer for one SILT command. That command may, of course, be a block, so the "with" can extend over any number of statements. In the example in section 5.4, the first and last boxes are placed in poly. If a box call has a layer specification, it holds only for that box. "With" commands can be nested, with the following results:

```
with poly do
    begin
    place box(1,1) at (2,2);          (* set in polysilicon *)
    with metal do
        begin
        place box(1,1) at (3,5);      (* set in metal *)
        place box(1,2,diff) at (2,3); (* set in diffusion *)
        place box(1,1) at (10, 11);   (* set in metal *)
        end;
    place box(5, 9) at (19, 20)       (* set in polysilicon *)
    end;
```

Allowable layers for the "with" command include: "buried" (buried contact), "contact" (contact cut), "diff" (diffusion), "implant" (implant), "metal" (metal) and "poly" (polysilicon).

5.8 Conditional Commands

A SILT conditional command is a simple "if-then" statement. It takes the general form:

```
if <boolean expression> then <SILT command>;
```

The boolean expression is evaluated, and the SILT command is expanded if it is true. There is no "else" clause -- use another conditional statement if this is necessary. This is not intended to be a heavily-used feature of SILT. It could be used to generate river-routing cells, for example, and to decide which way a wire bends.

5.9 The CIF_List Command

The "CIF_List" command attaches a name to a point (and optionally to a layer as well) in the CIF file generated. The CIF generated by this command is not standard CIF -- it is the "94" user extension used at Stanford and some other sites. The text is listed in the plot, and at Stanford, at least, it is required to be a single identifier (no spaces allowed). Examples of two typical CIF_List commands follow:

```
cif_list "text" at (9, 10);
cif_list "text1" at (10, 11, metal);
```

5.10 External Declarations

If libraries of symbols in CIF form are available, it is possible to make use of the symbols contained in them in SILT using the "extern" command. The symbol that is declared external is assumed to be absolute, and information about its minimum and maximum x- and y-values is not available. If SILT encounters an Extern statement, an entry is made in a symbol file that contains the CIF number used internally, the symbol name (used in the CIF file), and the file name. Extern declarations should be intermixed with the rest of the symbol declarations in the SILT file. A typical extern declaration appears below:

```
extern cifname "filename";
```

The information in the symbol file can be used to link together SILT and CIF files, either manually, or with a program. At present, there is a linker that can link a single SILT file to any number of ICARUS-produced CIF files.

5.11 Assignment Statements

There are a number of allowable types of assignment statements allowed in SILT. The right-hand-side is evaluated, and is assigned to the variable on the left-hand-side. An error occurs if the two sides do not conform. Any variable that is stored as a single real number conforms to any other, any signal conforms to any other, and any signal vector of length *n* corresponds to any other of the same length, and so on. Following are a few examples of typical SILT assignment statements:

```
a := 5;
sig.x := 7 + a;
sig1 := sig2;
sigvec<0:3> := sigvec1<4:7>;
sig1 := metal;
sig1<2>.y := 15;
sigvec<0:7> := inst.sigout<0:7>;
sig<5> := (1,2,metal), (3, 4), (5, 6, diff);
```

All the examples above except for the last should be clear. In the last case, the fifth signal in the vector "sig" is assigned a series of point-layer combinations. These are added to any sets of values the signal may already have. If an assignment is made to a signal suffixed with a ".x" or ".y", or a layer assignment is made, then if the signal has a point defined, its corresponding value will be replaced. If it has no value, a new slot is made. Thus, one can put in a new signal value as follows:

```
sig.x := 5;
sig.y := 7;
sig := poly;
```

But if the command:

```
sig.x := 6;
```

is given, a new signal point is not begun -- the value 5 is simply written over.

Assignments of the form:

```
sig := (3,4,poly);
```

always generate a new point instance.

It is legal, although probably bad practice, to re-use variable names as illustrated in the following example:

```
i := 7;
place box(i,i) at (0,0);
i := 9;
place box(i,i) at (10,10);
```

If assignments occur only once to each variable, the language becomes declarative. Future versions of SILT may print warnings when a variable (other than an iteration variable, of course) is assigned to more than once.

5.12 Connect Commands

The SILT connect commands are used to make sure that a signal point placed by one symbol call coincides with a similar point in another symbol call. If a connect command is given, SILT simply makes sure that the points in question do coincide. If not, an error message is generated, and SILT continues to expand the file. Remember that a signal can correspond to many points. If two signals are connected (that is, an appropriate connect command appears in the SILT file), the connection is

Page 24 blank

considered to be successful if any point from one coincides with any point from the other. If layers are specified as part of the point, then the layers must conform as well. SILT allows certain combinations of layers for this purpose. metal_poly, for example, means that this point can be connected to another point of type metal or poly.

Following are some simple examples of the use of the connect command:

```
connect a to b;
connect a.sigout to b.sigout;
for i := 1 to 7 do
    for j := 0 to 7 do
        connect a[i].sigout<j> to a[i-1].sigin<j>;
```

In addition to the simple connect command illustrated above that connects a signal to a signal, there is a sometimes more convenient form that makes sure that all the signals having the same name in two instances are connected. This is useful if a bus passes through, and one would like to make sure that all the bus signals are connected. Some examples follow:

```
connect all inst1 to inst2;
for i := 1 to 7 do
    connect all inst[i] to inst[i-1];
```

5.13 The Route Command

The route command implements a simple river-router to connect one signal vector of points to another. The points in the signals must include a layer chosen from {diff, poly, metal}. Both signal vectors must be essentially parallel -- i.e. they must both be monotonic in the x-coordinates or both in the y-coordinates. Some examples of the route command appear below:

```
route(sig1<0:7>, sig2<0:7>, 3, ud);
route(sig1<0:7>, sig2<8:15>, 4, lr);
```

In the first example, sig1<0> is routed to sig2<0>, and so on; in the second example, sig1<0> is routed to sig2<8>. In the first example, the widths of all the routing wires are 3, and in the second example, 4. The first example routes the wires generally up-down (the signal vectors are parallel to the x-axis), and in the second example, the routing is from right to left. The endpoints of the wires do not have to lie in a line. The points in the signal vectors mark the centers of the endpoints of the wires, and the wire layer is determined by the layers of the endpoints. There is an example of the use of the route command in I.

I. Examples

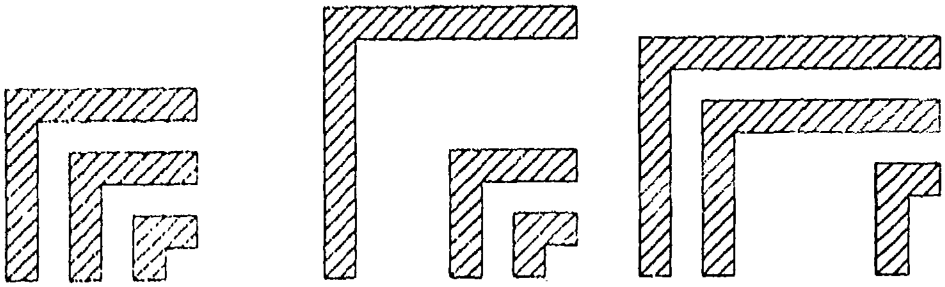
Three examples appear below. Some of them are not done in the most efficient way, but are done in a way that illustrates as many features of SILT as possible. All the examples are probably too small to be realistic.

The first one is extremely simple, and is made up of a few calls on a symbol that consists of a few boxes. It is intended to illustrate relative geometry. The second example illustrates the use of the route command. It also illustrates a few other features of SILT.

The third is the inverter from the Mead-Conway text [3] with some stretch built into it. In this example, the metal-metal distance can be altered, the input and output wires can be shifted up and down, and the pulldown ratio can be altered. Four different cell configurations are illustrated.

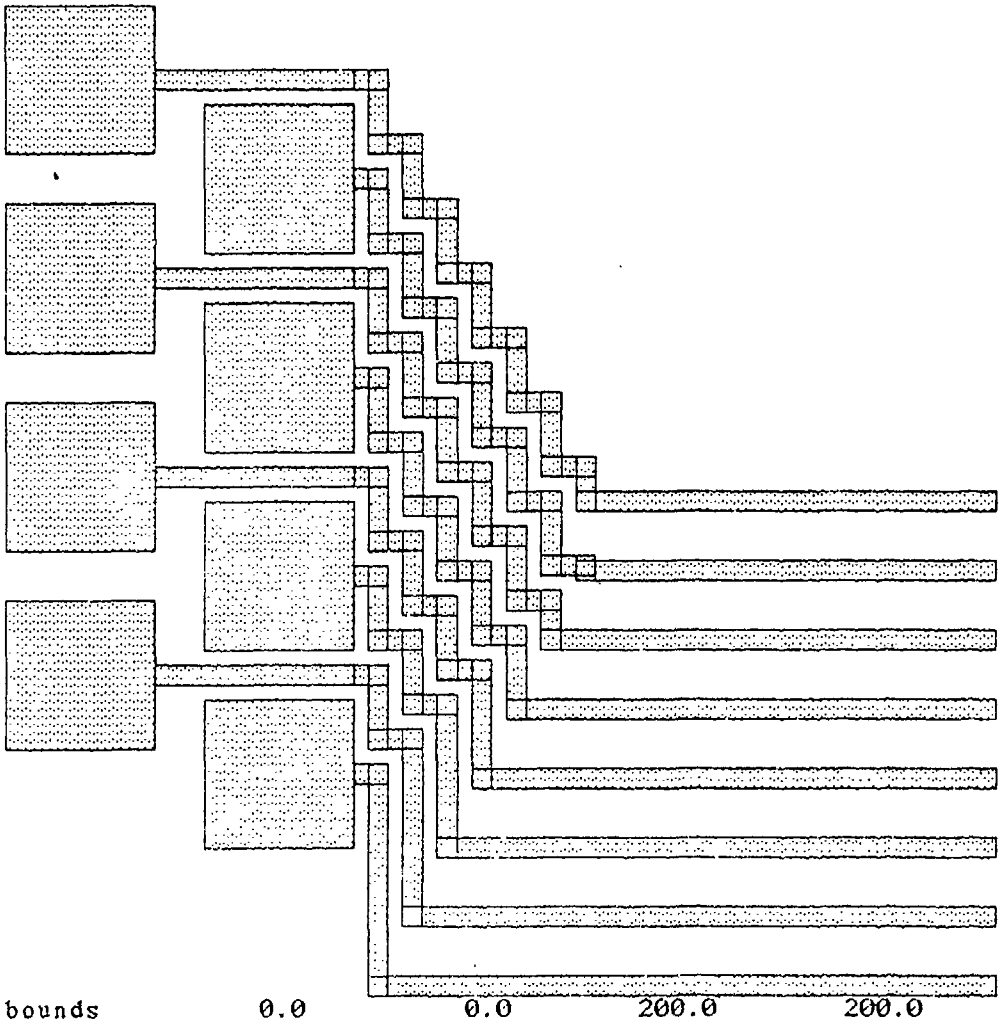
```
file relative_geometry;
symbol three_bend(xvar x1, x2, x3; yvar y1, y2, y3);
default x1 := 4; x2 := x1 + 4; x3 := x2 + 4;
        y1 := 2; y2 := y1 + 4; y3 := y2 + 4;
    begin
    with poly do
        begin
        place box(2, y3+2) at (0, 0);
        place box(2, y2+2) at (x1, 0);
        place box(2, y1+2) at (x2, 0);
        place box(x3 - x2 - 2, 2) at (x2+2, y1);
        place box(x3 - x1 - 2, 2) at (x1+2, y2);
        place box(x3 - 2, 2) at (2, y3)
        end;
    end;
begin
place three_bend at (0,0);
place three_bend(x1 = 8, y3 = 15) at (20, 0);
place three_bend(x2 = 16, y1 = 5) at (40, 0)
end.
```

Figure 5-1: Relative Geometry Example



```
file pad_router;
constant pad_size := 30;
scalar i;
signal wire_target<0:7>;
symbol pad();
signal output;
export output;
    begin
    place box(pad_size, pad_size, metal) at (0,0);
    output := (pad_size/2, pad_size, metal)
    end;
symbol pad_raft(scalar pad_spacing);
signal pad_outputs<0:7>;
scalar i, pad_sep;
export pad_outputs;
    begin
    pad_sep := pad_size + pad_spacing;
    for i := 0 to 3 do
        begin
        lower[i]:: place pad at (i*(pad_sep), 0);
        upper[i]:: place pad at (pad_sep/2+i*(pad_sep),
                                 pad_sep);
        end;
    for i := 0 to 7 do
        begin
        if (i mod 2) = 0 then pad_outputs<i> := lower[i div 2].output;
        if (i mod 2) = 1 then pad_outputs<i> := upper[i div 2].output
        end;
    end;
begin
raft:: place pad_raft(pad_spacing = 10) at (0,0);
for i := 0 to 7 do
    wire_target<i> := (100 + i*14, 200, metal);
route(wire_target<0:7>, raft.pad_outputs<0:7>, 4, ud);
end.
```

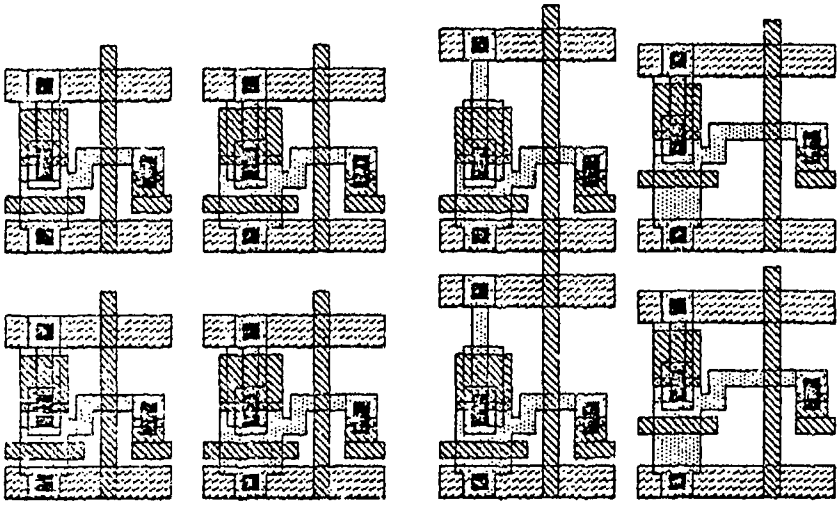
Figure 5-2: River Routing Example



```
bounds          0.0        0.0      200.0      200.0
CIF bounds          0.0        0.0      200.0      200.0
scale 40
```

```
file m_c_inverter;
symbol inv(xvar x1; yvar y1, y2; scalar p_width);
xvar diff_ctr, diff_edge;
default y1 := 5; y2 := y1 + 14;
        p_width := 6; x1 := 12 + p_width - 6;
constraint p_width >= 6; x1 >= 6 + p_width;
           y1 >= 5; y2 >= y1 + 14;
    begin
    diff_ctr := 2 + p_width/2;
    diff_edge := 2 + p_width;
    place butt rotated 9 at (diff_ctr+2, y1+3);
    place butt rotated 3 at (x1+4, y1+8);
    with diff do
        begin
        place cut at (diff_ctr -2, 0);
        place cut at (diff_ctr - 2, y2);
        place box (p_width, y1+4) at (2,3);
        place box (1,2) at (diff_edge, y1+3);
        place box (2,5) at (diff_edge+1, y1+3);
        place box (x1-diff_edge+1, 2) at (diff_edge+3, y1+6);
        place box (2, y2-y1-7) at (diff_ctr-1, y1+7)
        end;
    with poly do
        begin
        place box (p_width, 7) at (2, y1+6);
        place box (4+p_width, 2) at (0, y1);
        place box (2, y2+7) at (x1, 0);
        place box (5, 2) at (x1+4, y1)
        end;
    with metal do
        begin
        place box (x1+9, 4) at (0,0);
        place box (x1+9, 4) at (0, y2)
        end;
    place box(5, 10, implant) at (diff_ctr-2.5, y1+4.5)
    end;
symbol inv_set();
    begin
    place inv() at (0,0);
    place inv(p_width = 8) at (25,0);
    place inv(p_width = 7, y2=24) at (55,0);
    place inv(x1=16, y1=8) at (80, 0)
    end;
begin
inst:: place inv_set() rotated 3 at (0,0);
place inv_set() rotated 3 at (inst.xmax, 0);
end.
```

Figure 5-3: Inverter Example



II. Syntax

```
expression -> term;
    -> expression '+' term;
    -> expression '-' term;

term -> factor;
     -> term '/' factor;
     -> term '*' factor;
     -> term 'div' factor;
     -> term 'mod' factor;

factor -> unsigned_factor;
       -> '-' unsigned_factor;

unsigned_factor -> 'number';
                -> '(' expression ')';
                -> value;
                -> 'bitop' '(' expression ',' expression ')';

boolexp -> boolterm;
        -> boolexp 'or' boolterm;

boolterm -> boolfactor;
         -> boolterm 'and' boolfactor;

boolfactor -> primboolfactor;
           -> 'not' primboolfactor;

primboolfactor -> '(' boolexp ')';
               -> constraint;

instance -> instance_name;
         -> instance_name instance_qualifier;

simple_variable -> variable_name;
                -> variable_name signal_coordinate;

suffixed_variable -> simple_variable suffix;

suffix -> '.x';
       -> '.y';

instance_name -> 'ident';

variable_name -> 'ident';

value -> instance '.' simple_variable;
      -> simple_variable;
      -> instance '.' suffixed_variable;
      -> suffixed_variable;

signal_coordinate -> '<' expression '>';
```

```
signal_range -> '<' expression ':' expression '>';

signal_vector -> variable_name signal_range;

instance_qualifier -> '[' expression ']';
                   -> '[' expression ',' expression ']';

symbol_definition -> 'symbol' 'ident' formal_parameter_list ':'
                     symbol_head
                     'begin'
                     symbol_tail
                     'end' ';';
                  -> 'extern' 'ident' 'string' ';';
formal_parameter_list -> '(' variable_list ')';
                      -> '(' ')';
                      -> ;

symbol_head -> constant_declaration
               variable_declaration
               export_declaration
               default_declaration
               constraint_declaration
               symbol_definition_list;

symbol_definition_list -> symbol_definition_list symbol_definition;
                       -> ;

constraint_declaration -> 'constraint' constraint_list ';';
                       -> ;

constraint_list -> boolexp;
                -> constraint_list ';' boolexp;

constraint -> expression '<=' expression;
           -> expression '>=' expression;
           -> expression '<>' expression;
           -> expression '=' expression;

default_declaration -> 'default' default_list ';';
                    -> ;

constant_declaration -> 'constant' constant_list ';';
                     -> ;

constant_list -> constnt;
              -> constant_list ';' constnt;

constnt -> 'ident' ':=' expression;

default_list -> default;
             -> default_list ';' default;

default -> simple_variable ':=' expression;
        -> suffixed_variable ':=' expression;
```

```
variable_declaration -> variable_list ';';
                     -> ;

export_declaration -> 'export' ident_list ';';
                   -> ;

variable_list -> variable_type_list;
              -> variable_list ';' variable_type_list;

variable_type_list -> 'xvar' ident_list;
                   -> 'yvar' ident_list;
                   -> 'scalar' ident_list;
                   -> 'signal' signal_list;

signal -> 'ident';
       -> 'ident' signal_range;

signal_list -> signal;
            -> signal_list ',' signal;

ident_list -> 'ident';
           -> ident_list ',' 'ident';

symbol_tail -> symbol_command;
            -> symbol_command ';' symbol_tail;

silt_file -> 'file' 'ident' ';'
             symbol_head
             'begin'
             symbol_tail
             'end' '.';

symbol_command -> symbol_label '::' unlabeled_command;
               -> unlabeled_command;
               -> box_call;
               -> box_to_call;
               -> butt_call;
               -> cut_call;
               -> connect_command;
               -> assignment;
               -> with_command;
               -> iterate_command;
               -> cif_list_command;
               -> block_command;
               -> conditional_command;
               -> route_command;
               -> array_command;
               -> ;

route_command -> 'route' route_list;

route_list -> '(' vectored_signal ',' vectored_signal ',' expression ')';
           -> '(' vectored_signal ',' vectored_signal ','
                  expression ',' route_direction ')';
```

```
route_direction -> 'ud';
                -> 'rl';
                -> 'lr';

vectored_signal -> signal_vector;
                -> instance '.' signal_vector;

cif_list_command -> 'cif_list' 'string' 'at' position;

unlabeled_command -> symbol_call;

block_command -> 'begin'
                 symbol_tail
                 'end';

symbol_call -> 'place' 'ident' actual_parameter_list
               orientation_specification;
            -> 'ident' actual_parameter_list
               orientation_specification;

orientation_specification -> transformation_list;
                          -> ;

transformation_list -> transformation;
                    -> transformation ',' transformation_list;
                    -> transformation transformation_list;

actual_parameter_list -> '(' key_call_list ')';
                      -> '(' position_call_list ')';
                      -> '(' ')';
                      -> ;

key_call_list -> key_call;
              -> key_call_list ',' key_call;

position_call_list -> expression;
                   -> position_call_list ',' expression;

key_call -> simple_variable '=' expression;
         -> signal_vector '=' signal_vector;
         -> signal_vector '=' instance '.' signal_vector;

position -> '(' expression ',' expression ')';
         -> '(' expression ',' expression ',' layer ')';

transformation -> 'flipped' 'ud';
               -> 'flipped' 'rl';
               -> 'flipped' 'lr';
               -> 'rotated' 'number';
               -> 'at' position;

box_call -> 'place' 'box' box_parameters 'at' position;
         -> 'box' box_parameters 'at' position;
```

```
box_to_call -> 'place' 'box' box_parameters 'to' position;
            -> 'box' box_parameters 'to' position;

box_parameters -> '(' box_size ')';
               -> '(' box_size ',' layer ')';

box_size -> expression ',' expression;

butt_call -> 'place' 'butt' orientation_specification;
          -> 'place' 'butt' '(' ')' orientation_specification;
          -> 'butt' orientation_specification;
          -> 'butt' '(' ')' orientation_specification;

cut_call -> 'place' 'cut' orientation_specification;
         -> 'place' 'cut' '(' ')' orientation_specification;
         -> 'place' 'cut' layer_parameter orientation_specification;
         -> 'cut' orientation_specification;
         -> 'cut' '(' ')' orientation_specification;
         -> 'cut' layer_parameter orientation_specification;

layer_parameter -> '(' 'diff' ')';
                -> '(' 'poly' ')';

layer -> 'poly';
      -> 'metal';
      -> 'diff';
      -> 'buried';
      -> 'implant';
      -> 'contact';
      -> 'metal_poly';
      -> 'diff_poly';
      -> 'diff_metal';
      -> 'none';

assignment -> simple_variable ':=' expression;
           -> suffixed_variable ':=' expression;
           -> simple_variable ':=' layer;
           -> simple_variable ':=' position_list;
           -> signal_vector ':=' signal_vector;
           -> signal_vector ':=' instance '.' signal_vector;

position_list -> position;
              -> position_list ',' position;

connect_command -> 'connect' value 'to' value;
                -> 'connect' 'all' instance 'to' instance;

with_command -> 'with' layer 'do' block_command;

iterate_command -> 'for' 'ident' ':=' expression 'to'
                   expression 'do' symbol_command;

conditional_command -> 'if' boolexp 'then' symbol_command;

symbol_label -> instance;
```

```
array_command -> 'array' 'ident' a_list 'of' a_list_call
                 spacing 'at' position;
              -> 'place' 'array' 'ident' a_list 'of' a_list_call
                 spacing 'at' position;

a_list -> '[' expression '..' expression ']';
       -> '[' expression '..' expression ','
              expression '..' expression ']';

a_list_call -> 'ident' actual_parameter_list;

spacing -> 'spaced' position;
        -> ;
```

III. SILT Reserved Words

The following identifiers are reserved by SILT. The only ones that may be a little surprising are "x" and "y". These are used as suffixes of points.

```
ALL         AND         ARRAY       AT          BEGIN
BITOP       BOX         BURIED      BUTT        CIF_LIST
CONNECT     CONSTANT    CONSTRAINT  CONTACT     CUT
DEFAULT     DIFF        DIFF_METAL  DIFF_POLY   DIV
DO          END         EXPORT      EXTERN      FILE
FLIPPED     FOR         GLASS       IF          IMPLANT
LR          METAL       METAL_POLY  MOD         NONE
NOT         OF          OR          PLACE       POLY
RL          ROTATED     SCALAR      SIGNAL      SPACING
SYMBOL      THEN        TO          UD          WITH
X           XMAX        XMIN        XVAR        Y
YMAX        YMIN        YVAR
```

References

1. Davis, Tom, and Clark, Jim. YALE User's Guide: A SILT-Based VLSI Layout Editor. Stanford, 1982.

2. Hennessy, John. The Stanford Parser Generator. Stanford, 1980.

3. Mead, Carver, and Conway, Lynn. *Introduction to VLSI Systems.* Addison Wesley, 1980.

4. Newell, Martin. The ALE Graphics Editor. XEROX PARC

5. Saxe, Tim. CLL - A Chip Layout Language (Version 2). Used in the Stanford VLSI design class

YALE User's Guide:

A SILT-Based VLSI Layout Editor

Tom Davis and Jim Clark

Technical Report No. 233

October 1982

This research was supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680.

YALE User's Guide:

A SILT-Based VLSI Layout Editor

Tom Davis and Jim Clark

Technical Report No. 233

October 1982

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

YALE is a layout editor which runs on SUN workstations, and deals with cells expressed in the SILT language. It provides graphical hooks into many features describable in SILT. YALE runs under the V Kernel, and makes use of a window manager that provides a multiple viewport capability.

Key Words and Phrases: VLSI, design tools, layout editors, relative geometry

Table of Contents

1. Introduction

The YALE (Yet Another Layout Editor) layout editor makes hierarchical cell layouts on a SUN workstation. It is meant to be used with the SILT translator. All the files produced and read by YALE are written in the SILT language (see [3]). The SILT program translates these into a CIF format for use with other programs such as design rule checkers, circuit simulators, and mask-making software.

1.1 System Overview

The YALE layout editor runs under the V Kernel (see [2]) on a SUN workstation. The V Kernel is a message-based kernel supporting multiple processes of which YALE might be only one. Since the current implementation of YALE is large, it is unlikely that too much else will be running on the workstation at the same time as YALE.

To run YALE with any reasonably large layout at all requires more than the minimal 256K SUN configuration. The more memory is available on the SUN workstation, the better YALE will work.

Even if YALE is the only program running under the V Kernel, there are two processes with which you, as the user, must be familiar. One is the layout editor itself, and the other is the window manager. YALE itself has no idea where its viewports are presented on the screen, or even how many viewports there are. You are free to create more viewports opening on different parts of the cell being currently edited, and to move these viewports around and to adjust their sizes and magnifications. There is nothing special about YALE in this respect. Any process using the above-mentioned window manager can do the same thing.

At any time during a session, all the keystrokes and mouse-clicks are directed either to the window manager or to the YALE editor (assuming these are the only two processes running). Obviously, the input is interpreted differently by the two processes, so if something surprising happens, make sure that your input is going to the process you think it is.

It is easy to tell at a glance whether you are typing to (or "mousing" to) YALE or to the window manager. The shape of the cursor changes. In both cases, the cursor is an upward-pointing arrow, but for the window manager, it is shorter, and its lower half forms the letter "w".

There are really more than two processes running during a YALE session, but the editor and the window manager are the only two with which you need to deal. Other processes include one process that is busy watching the keyboard for input, and another that always watches the mouse.

1.2 Starting Up

To load YALE, you must use the V Kernel loader, called "Vload". This is done by typing "n Vload" (note the capitalization here) to the monitor of the SUN workstation. After a while, this program will respond with something like:

```
V Kernel Loader - Version 3.1 - 28 June 1982

Program name:
```

Type in "Yale" (again, note the capitalization), and press <CR> (<CR> stands for the carriage return key). The system will respond by typing a few lines of exclamation marks, and finally with:

```
>
```

Type "c<CR>" ("c" stands for continue -- the reason that this step is necessary is that breakpoints could be put in here), and the program will start.

Three initial viewports will be painted on the screen whose functions are described in the next chapter, and you will be prompted for some startup information in the tty viewport (which is positioned initially in the lower-left hand corner of the screen).

1.3 Implementation

YALE is implemented entirely in the "C" programming language, and most of the internals of the YALE editor (almost everything save the display code) were thoroughly tested on a VAX before being ported to the SUN workstation. Since most of it has already been ported once, it should not be too difficult to port it again. All that needs to be written is another set of display routines.

Although YALE runs under the V Kernel, it does not make heavy use of its services. The V Kernel provides a process to track the cursor, and to read the mouse and keyboard. The window manager runs as a separate process, but this is not necessary, and in the initial implementation, it was part of the YALE editor.

A reasonably clean separation of the editor part from the window manager part has been made to simplify some experiments using YALE in a distributed mode -- YALE will run on a VAX, and the window manager and a display list interpreter will run on the SUN workstation.

In addition to the V Kernel, YALE makes use of the SUN rasterop (see [1]) package for the display of rectangles and stipples on the screen, and of the leaf package (see [4]) for remote file access.

2. User Interface

This chapter describes the general features of the user interface for both YALE and the window manager. An attempt has been made to make the command interaction similar for both programs. Most of the conventions described below apply both to the layout editor and to the window manager.

2.1 Initial Viewports

When YALE starts up, it displays three viewports. One of these is called the **tty viewport**, and is used primarily for command feedback, error messages, and user type-in. It behaves exactly like a glass teletype in that each line is typed at the bottom, and lines typed earlier are scrolled up. Although the initial viewport is small, the tty window keeps 24 lines of text, and earlier commands can be examined by calling on the window manager and enlarging the viewport.

The second viewport is called the **status viewport**, and contains some YALE status information. Such things as the current file, the name of the cell being edited, x- and y- coordinates of the last mouse click, the currently selected layer, and the default widths of the layers are presented. The information here will be covered in more detail later. See .

The third viewport is the main YALE **graphics viewport**. This is where the currently open cell is presented. Most other YALE viewports will be graphics viewports, but views opening on different portions of the cell being edited.

2.2 The Mouse

YALE and the window manager both receive most of their commands from the mouse. When the mouse is held with its three buttons on top, the left-most mouse button is called number 1, the center is number 2, and the right-most button is numbered 3. In some of the prompting that appears on the screen, and in the documentation that follows, they are referred to as MB1, MB2, and MB3.

Some YALE commands require that more than one mouse button be pressed at the same time. Since it is impossible to press the buttons at *exactly* the same time, the mouse input is interpreted by YALE as follows: The interpretation begins when the transition is made from all buttons up to at least one button down. It ends when all buttons are up again. All buttons depressed in the interim are recorded as part of the mouse event. Thus, as long as you have not released all the buttons, you can always press another button. Because of this, if the mouse event is not yet complete, it is always possible to press down all three buttons, and YALE takes advantage of this by defining a three-button push as aborting the command. If you accidentally press the wrong button and notice it before it is released, simply press dowr the other two buttons and then release all three, and there will be no net effect.

In both YALE and the window manager, the general philosophy is to bind the most useful commands to the left and center mouse buttons (MB1 and MB2). All other commands are accessed by one or more pop up menus, described in the next section. For both programs, the pop up menu containing the rest of the commands is accessed by pressing and releasing the third mouse button.

The action of pointing to an object or screen position with the cursor, and choosing it with a mouse button click is often referred to here as "bugging". One can thus "bug a rectangle" to select it, or identify a viewport to move by "bugging it".

2.3 Pop Up Menus

Since there are only 7 mouse button combinations, even if they all were to be used, there are too many YALE commands to go around. The same thing is true of the window manager, so only the most useful commands are bound directly to mouse clicks, and the rest of the commands are invoked using pop up menus. For both the window manager and for general YALE layout editor commands, the pop up menu is gotten by using MB3 by itself. A menu containing a variable number of items appears under the cursor at that point. To select a command presented there, move the mouse until the tip of the cursor inside its box, and press any button. (In other words, "bug" the correct menu entry.) Sometimes menus are two-level, so a second menu will appear for the sub-command. If you invoke a menu by accident and do not really wish to select any of the commands in the menu, simply move the cursor completely outside the menu and press any button.

Important! To get a pop up menu for YALE, you must press the third button while the cursor is in one of YALE's windows. In this way, it is possible to use the window manager with more than one process. If the third mouse button is pressed while it is outside any window, it will have no effect. On the other hand, when your input is directed to the window manager, the main pop up menu (for the window manager) can be pressed at any time.

2.4 Command Feedback

As each command is issued, whether it is clicked in with the mouse buttons, or accessed through a series of one or more pop up menus, an English sentence is gradually built up in the tty viewport both to show what command is being specified, and, where possible, the next input required of the user. If there is ever any confusion about which command is being specified, the last line in the tty viewport shows what is going on and what is expected next.

There are a few abbreviations that are commonly used in this feedback in addition to MB1, MB2, and MB3 for the mouse buttons. These include:

Abbreviation	Meaning
T:	Type in textual data, followed by <CR>.
B:	"Bug" an object or position on the screen by pointing to it with the cursor and pressing (usually) MB1.
M:	A menu selection of some sort is to be made. Move the cursor so that its tip is in the correct menu entry, and press a mouse button. Some menus are special, and have "typein" entry at the bottom. If this entry is selected, you will be asked to type in the correct information, followed by a <CR>. This mechanism is used when there is a set of common choices, but where you may wish to use some unusual choice from time to time.

2.5 Typing in Information

From time to time, certain of the commands require that you type in some textual information -- the name of a new cell definition, the name of a reference point, or the name of a file to be used for input or output. When this happens, a prompt appears in the command feedback viewport, and you simply type in the text, followed by a carriage return. For new names, YALE obviously has no choice but to ask you directly for the information.

On the other hand, since the most commonly typed text will be the names of cell definitions that are already known (such as "expand cell named ...", "create instance of cell named ..."), YALE keeps a list of a few of the most recently referenced cell names. Thus, when you need to specify a cell name, YALE puts up a Pop Up menu containing these most recently referenced cells. The last item in the menu is always "Typein", and if that entry is selected, you will be requested to type in the name of the definition as you would for any other text string.

YALE automatically converts all typed input and all input from files to lower case (except, of course, for UNIX file names). You may type your names in using any case you wish, but YALE will print them back to you in lower case only.

Usually, a mechanism analogous to the three-mouse-button abort exists for textual input. When YALE requests type-in, it is not looking at the mouse, so it cannot interpret a three-button abort. When a name of some sort is required and you have committed to a typein, typing a <CR> with no text aborts the command. This mechanism is not so universal as is the one using three mouse buttons.

2.6 The YALE Coordinate System

YALE (and SILT) work in a standard mathematical right-handed cartesian x-y coordinate system. SILT allows arbitrary real coordinates, but YALE restricts the coordinates to be integers or half-integers. When selecting a point on the screen, the nearest point with half-integer coordinates is chosen.

The grid marks that can appear on the screen are always placed at distances that are equal of a power of 2 times lambda. The exponent can be negative, however.

3. The Window Manager

3.1 Displays and Viewports

During any session, there may be many distinct "universes" whose contents you may wish to view. When YALE starts up, there are three -- the text shown in the tty viewport, the information shown in the YALE status viewport, and the geometry of the cell being edited in the graphics viewport. If the terminal is being used for other user processes, additional viewports may be required.

Each of these "universes" is called a **display**. A process may have zero or more displays associated with it. (In particular, the mouse and keyboard watcher processes have zero.) In addition, it may be useful to have more than one view of any given display. For VLSI editing, one ofter wants to have one large-scale view of the cell being edited, and another zoomed-in view of the small region being actively changed. For complex, long-distance wiring, it may be useful to have views of each of the areas on the chip where the wires bend.

It is sometimes useful to make a distinction among the terms "display", "viewport", and "window", especially between the latter two. The display contains all the objects in the universe of interest, whether they are shown on the screen or not. The viewport is the physical area on the display screen upon which objects are displayed, and the window contains the same information as the viewport, but is described in terms of world (or YALE) coordinates instead of in screen coordinates. Thus, one might move a viewport on the display screen, and one would center a window coordinate in the current viewport.

Each individual view of a display is called a **viewport**, and occupies some physical space on the screen. The window manager can change both the part of the display being viewed, and that view's magnification.

The window manager can handle up to 8 different viewports. All must be rectangular and non-overlapping. It is possible to shrink little-used viewports, so it is not impossible to make use of all 8 viewports occasionally. It can also be used to create more or fewer viewports associated with the same display. Usually, however, 4 or 5 viewports should be all that are required. If the window manager is someday modified so that it can handle overlapping viewports, all this may change.

3.2 Entering and Exiting the Window Manager

When YALE starts running, all input is directed to the YALE layout editor. There are two ways to get the attention of the window manager, and to begin directing input to it.

The easiest method is via a selection from YALE's main pop up menu. Press MB3, move the cursor to point at the entry entitled "Window Manager", and bug it. The cursor will change from a simple arrow to a shorter arrow with the character "w" underneath.

The second method is to type the so-called "brain escape character", which is initially set to be "↑C" (control-C). The cursor will immediately change to its window manager form. The brain escape character can be reset to something besides "↑C" using a command to the window manager (see 3.4).

The window manager is exited in only one way -- this is via a selection from the window manager's main menu (again accessed using MB3). The menu entry is labeled "Return to YALE".

3.3 Zooming In and Out

The most useful commands that can be issued to the window manager are for zooming in and out on the view. MB1 is bound to "Zoom In" and MB2 is bound to "Zoom Out". These commands affect only the viewport in which the cursor is displayed when the button is pressed. It is therefore possible to have different viewports displaying the cell being edited with different magnifications.

When MB1 ("Zoom In") is pressed in a viewport, the magnification in that viewport is doubled, and the point at the tip of the cursor arrow [illegible] appears in the center of the new screen. MB2 ("Zoom Out") has just the opposite effect -- the magnification factor is cut in half, and point at the tip of the cursor arrow again appears in the center of the new view.

Notice that one can change the view without changing magnification simply by zooming out, and then zooming in to a new center, and can thus be accomplished using two mouse clicks. If the cell being displayed is complicated, however, it is probably easier to use the "Center Window" command (described below), since the display is re-created after each zooming action.

In some viewports, zooming in and zooming out have no effect on the material displayed. This is usually the case where the viewport is displaying some textual material. Neither the tty viewport nor the status viewport are affected by these commands, but all the YALE graphics viewports are.

3.4 The Main Window Menu

The main window manager menu is accessed using MB3. Each of the possible menu commands is described in a paragraph below:

Create viewport. This command creates another viewport on the screen. The next two mouse clicks are interpreted as opposite corners for the new viewport, and may be given in any order. Finally, an existing viewport is bugged to show which display is to be painted in the newly created viewport. If an attempt is made to create a new viewport that overlaps existing viewports, the error message "illegal stretch" will appear in the tty viewport.

Move Edge. To change the shape of a viewport, bug the "Move Edge" entry in the main window manager menu, move the cursor to an edge or corner of a viewport, and press any mouse button. Then move the cursor to a new screen position, and press the button again (remember that if all three buttons are presed at any time, the command can be aborted with no effect). The viewport is redrawn with its corner or edge moved to the second mouse position. Again, since viewports are not allowed to overlap, a common error is "illegal stretch".

Move Viewport. To move a viewport rigidly on the screen so that it remains the same size and shape, and continues to view the same display, bug the "Move viewport" command in the main window manager menu. Next, choose a point of reference in the viewport to be moved, move the cursor to that point, and press a mouse button. When the cursor is moved to a new position on the screen, and a mouse button is pressed, the point of reference in the old viewport is moved to that position. Usually this command is easiest to control if the reference point is selected near one of its corners. The window manager makes sure that the viewport in its new position does not lie outside the screen boundaries, and that it does not overlap existing viewports. If an attempt is made to move a viewport off the visible screen, its position is adjusted to put it entirely on-screen. If an attempt is made to overlap another viewport, an error message is printed in the tty viewport, and nothing happens.

Delete Viewport. This command deletes the viewport in which the cursor sits when the next mouse button is pressed. *Don't delete the last viewport on any given display, or you may have a hard time getting back the view.* This mis-feature should be corrected someday.

Center Window. This command allows one to change the view so that a new point in the display is moved to the center of the viewport. Simply press a mouse button with the cursor in a viewport, and the viewport will be redrawn with that point moved to the center of the new view. It was pointed out earlier that exactly the same change can be affected using a "Zoom Out" followed by a "Zoom In" command.

Redraw. From time to time, YALE or the window manager gets mixed up about exactly what is on the screen. This command simply causes the contents of all viewports of be redrawn from scratch. When all the bugs are removed, this command will be, too.

Brain Escape. This command resets the window manager escape character (which was initially set to be ↑C). A new escape character is typed, followed by a carriage return. The only restriction is that the character can not be "%" (See 11.1), but it is probably a bad idea to use printing characters that you may wish to use for typing information to YALE. Since YALE has a built-in command for accessing the window manager, this command probably has low utility. Other programs that will eventually run under the V Kernel may have no idea that there is anything like a window manager, and it is for them that this facility was included.

Toggle Grid. To aid in editing, it is sometimes useful to have a grid presented on the screen. The grid points are always the same distance apart on the screen, so at different magnifications, they correspond to different lambda-measures. After selecting the "Toggle Grid" option, you must then show which viewport is affected with another mouse click. If the grid is currently displayed in that viewport, it is turned off, and vice-versa. The grid may be toggled in any viewport, but it is probably not too useful to display a grid in any but a graphical viewport.

Return to Yale. This command diverts all further type-in and mouse clicks back to the YALE layout editor.

4. Editing with YALE

The next few chapters deal with the YALE layout editor itself. This first chapter gives an overview of the editing system, and the others describe in detail the commands that can be issued.

4.1 Yale Overview

Before going into a description of the meaning of each of the YALE editing commands, a short description will be given of the various sorts of things that the YALE editor manipulates. What is presented in the next few paragraphs is condensed, and an understanding of the SILT language would be extremely useful. SILT is a powerful language for describing cell layout, and YALE deals only with a small subset of the available SILT commands.

Because of this, it may be necessary to use a combination of YALE and hand-editing of the SILT format files produced by YALE to deal with a complicated layout. All the basic cell layout can be done with YALE, together with some hierarchical depth. If it is necessary to make use of SILT's river-routing facilities, or of other higher-level features, they will have to be specified by hand, in SILT, using the text editor of your choice.

Each layout is defined in terms of a hierarchical set of symbol calls[1], with one symbol designated as the master layout symbol. A general SILT file could have many different commands in the main file body, but YALE restricts it to having a single symbol call in the main file body.

Each symbol is made up of rectangles, reference points, and calls on other symbols. See the SILT documentation for a complete description of reference points.

The rectangles on the screen are represented with stipple patterns, the symbol calls are normally expanded, and the reference points are displayed as labeled horizontal and vertical lines. In addition to the user-defined reference points, another pair (the x- and y- origins) are also shown. In some ways, these origins behave exactly as the other reference points, and in other ways they do not. The main difference is that it is not possible to move the origin reference points. The origin of the cell is the point where the origin reference points cross.

[1] In this documentation, the term "symbol" and "cell" are used almost interchangeably. SILT uses the term "symbol", but the term "cell" is also widely used.

A symbol instance is placed with its origin relative to a horizontal and a vertical reference point. A rectangle has its top and bottom edge relative to horizontal reference points and its left and right edges relative to vertical reference points. For a symbol or rectangle edge to be placed "relative to a reference point" means that it remains a fixed distance from that reference point. If the reference point is moved, then all objects placed relative to it are moved as well. If both of a rectangle's vertical edges are relative to the same vertical reference point, and the reference point is moved, then the rectangle will be moved rigidly. If the edges are relative to different reference points, then the rectangle will stretch or shrink if one of its reference points is moved without moving the other.

In SILT, there is really nothing analogous to the origin reference points, but they are required for a reasonable graphical interface. The SILT code for a rectangle produced relative to the reference points named "xref" and "yref" would look something like this:

```
place box(xref + 3, yref - 6, metal) to (xref + 11, yref + 1);
```

while a rectangle placed relative to the origin reference points would generate the following SILT code:

```
place box(3, -6, metal) to (11, 1);
```

If "xref" and "yref" are moved so that they are coincident with the respective origins, the two lines of SILT code would represent the same box.

4.2 A Typical Editing Session

Before going into detail about the commands available in the YALE editor, it is useful to give an idea of how a typical editing session would be carried out. Let us assume that you wish to design a new cell, and already have available a small library of SILT cells in a file on your VAX.

After loading the V Kernel and the YALE program, some initialization is required. You will be asked for the name of the VAX with your files, your user name and password, and the name of the file that contains (or will contain) the layout produced. After YALE has verified that the VAX is up, that the user name and password are valid, and that the file exists (if the name is a new one, it is created), you are left in a state where the input goes to the layout editor.

From now on, the procedure is to open one of the symbols for editing, to make changes to it, and then to close it and open another symbol. Only the cell currently open can be modified. Modifications take the form of adding or deleting rectangles, sub cells, and reference points. If a cell is closed, then one of its subcells opened and modified, and the original cell opened again, all instances of the subcell in the original cell will show the modifications.

Cells are opened either with the "Expand Cell" command, or by creating them with the "Create Cell Instance" command. Cells are closed simply by opening another cell. Having a cell open is not really analogous to having a file open on a computer -- open cells are not particularly vulnerable. The fact that a cell is open simply means that it is the one displayed, and to which modifications will be made. There are not too many YALE operations that cannot easily be undone, except possibly deleting an entire cell definition, and deleting all the selected items, when many of them are selected.

When a new symbol is created, it contains nothing but an x-origin and a y-origin. These behave somewhat as default reference points. At any point during the editing, exactly one horizontal and one vertical reference point is selected, and the selected reference point is shown by being displayed with a bolder line. In a newly created cell, the x- and y-origins are the selected reference points. Any geometry (rectangles and symbol instances) which is entered is placed "relative to" the currently selected reference points -- that is to say, if the reference point is moved, the objects placed relative to it will also be rigidly moved.

Rectangles that are to remain rigid when the reference points are moved are easily inserted -- just make sure that the appropriate vertical and horizontal reference points are selected before inserting the rectangle. If it is required th**a**t a rectangle stretch or shrink when some reference points are moved (i. e. one edge is relative to one reference point, and the other edge is to be relative to another), the usual procedure is to make sure that one of the reference points is selected when the rectangle is inserted, and then to re-reference the other edge to a different reference point.

It is easy to see if all the internal symbols and rectangles are placed relative to the correct ,eference points -- simply move the reference points around a little, and make sure that the components move and stretch as they should. Any parts that do not can then be re-referenced.

As the editing proceeds, it is usual to make backup copies of the work from time to time. The first time the "Write Backup" command is invoked, you will be asked for a file name and a name of the master symbol. (The master symbol is usually the one corresponding to the entire chip. If there is no master symbol, any symbol name will do.' After this, making a backup is simple -- just issue the "Write Backup" command, and the old backup file will be over-written with the new version of the layout.

At the end of an editing session, the "Write Main File" command is usually given, followed by the "Quit" command. YALE is aborted, and you are returned to the SUN monitor.

4.3 The Status Viewport

Throughout a YALE editing session, one viewport is devoted to presenting the status of the session. The meaning of some of the information in this window is obvious, such as the name of the main input file and the name of the cell currently open for editing.

The most important feature of the status viewport is the set of seven small stipple pattern samples. These are the stipple patterns for the various layers. Above each sample of the pattern is an abbreviation for the name of the layer: "metl", "poly", "diff", "impl", "burd", "glas", and "cont" standing for "metal", "polysilicon", "diffusion", "implant", "buried", "glass", and "contact cut", respectively. Underneath each of these rectangles is a number indicating the default width of a wire made of this material. If the default width for the metal layer is 4, this means that whenever the metal layer is selected, all rectangles placed in the currently open cell will be made of metal, and will have width 4.

One of the seven patterns is selected (outlined), and this is the layer that will be used by the "Add Rectangle" command. To select another layer, simply move the cursor so that it points to a new stipple in the status viewport, and depress a mouse button. The newly selected stipple should be selected (outlined), and the previously selected layer should have its outline removed.

The default widths associated with each layer can also be changed. See 7.1, below.

Also in the status viewport are four entries labeled "x:", "y:", "dx:", and "dy:". Every time a mouse button is clicked in a YALE graphics viewport, these values are updated. The "x:" and "y:" values give the absolute position of the click relative to the origin of the cell; the "dx:" and "dy:" values give the displacement from the last "x:" and "y:" values. These entries can be used to measure distances within a cell, and to find out about where you are if the view is zoomed in so far that there are no reference points visible.

Finally, there is an entry for the currently active repeat command. A few of the common commands in YALE have the feature that although they must be specified using a series of menus originally, they may then be easily repeated. The name of the most recently issued command of this sort appears here.

To cause that command to repeat, simply depress the two left-most mouse buttons (MB1 and MB2). Exactly what happens next is slightly dependent on the repeated command. The three repeatable

Page 16 blank

commands are "Delete Selections", "Create Cell Instance", and "Add to Selections". See 8.2, 6.2, and 8.1, respectively for details on how the repeat command will behave in each circumstance.

5. Adding Rectangles and Selection

The addition of rectangles to a layout, and the selection of objects in a layout to modify are the two most common operations in YALE. Thus, each is tied to a single mouse click. The exact action of these two important commands is described in the two sections that follow.

5.1 Add Rectangle

At any point during the editing, there is a default layer for rectangles selected, and a default width for that layer. Placing rectangles on the screen involves marking the two endpoints of the wire using MB1. YALE figures out whether the selected points are more vertical or more horizontal and inserts an appropriate rectangle. The first point selected is guaranteed to be the center of one end of the wire. For example, suppose that MB1 is first pressed at the point having coordinates (15, 32), and is next pressed at (43, 35). The change in the x-direction (28) is much greater than the change in the y-direction (3), so the rectangle is assumed to be a horizontal one. Since MB1 was first pressed with a y-coordinate of 32, this will be the y-coordinate for the center of the inserted wire.

If MB1 is pressed accidentally, placement of the rectangle can be aborted by pressing any but MB1 (in particular, all three buttons can be pressed, causing the usual abort to occur).

When a rectangle is inserted, all currently selected items are de-selected, and the newly inserted rectangle is selected. This makes it easy to move it to the correct position or to delete it if it was placed incorrectly.

The left and right edges of the rectangle are placed relative to the currently selected vertical reference point, and the top and bottom edges are placed relative to the selected horizontal reference point. If there is no cell currently open, an error message will be presented in the tty viewport.

5.2 Select Item

To select an item, point the cursor at it, and press MB2. If the item is a symbol instance or reference point, it is simply selected. If it is a rectangle, things are a bit more complicated. If the cursor is in the center of the rectangle, the entire rectangle (all four edges) is selected. If the cursor is pointing to an edge, just that edge is selected. If the cursor is pointing to a corner, then both of the edges adjacent to that corner are selected. In every case, all other currently selected items are de-selected before the selection takes place. A selected symbol instance is displayed with a bold outline, and a selected rectangle edge is likewise highlighted.

Since objects may overlap each other in an arbitrary way, there must be some mechanism for disambiguating a selection that could be interpreted in more than one way. The disambiguation algorithm is described below.

1. If there is only one object under the cursor, then that object is selected.

2. If there are no objects under the cursor, then YALE looks within a few pixels of the cursor for nearby objects. If there is exactly one nearby object, then that is selected.

3. If there is more than one object under the cursor, or if there are no objects under the cursor, but there is more than one nearby object, then the areas of all possible objects are compared, and the object with the smallest area is selected. Reference points are judged to have essentially infinite areas.

4. If there is still ambiguity (i. e. there are still two or more items with exactly the same area and under the cursor),then the object with the smallest height-to-width ratio is selected.

5. Finally, if there is more than one object with exactly the same height and width of smallest area under the cursor, a series of possibilities is presented in the tty viewport. A short description of the object is given, and it can be selected by typing "y<CR>". Typing "n<CR>" causes the description of the next possible object to be presented. As soon as the user responds positively, that item is selected, and the selection is finished. If "n<CR>" is typed in response to every option, then nothing is selected.

Note that it is still possible to construct an example where it is impossible to select a certain piece of geometry. In particular, it will happen when a large object is completely covered by smaller objects, as is the metal layer in a butting contact. If this happens, the only way out is to delete or move the object(s) causing the conflict, select and deal with the object of interest, and then re-create or move back the other objects. This should not happen often.

Exactly the same disambiguation algorihm is used with the "Add Selection" command, described below.

The most commonly-used commands in YALE are the "Add Rectangle" and "Select" commands, accessed using MB1 and MB2. All the other YALE commands are invoked through a series of one or more pop up menus. The main pop up menu is accessed by pressing MB3. It presents 10 options that are described in the following chapters. Most of the items in this main menu are really just categories of commands, and simply bring up a sub-menu containing specific commands. A few of the entries, however, are bound directly to commonly-used commands.

6. Create commands

Selecting the "Create" entry from the main YALE pop up menu brings up the sub-menu of the create commands. There are five entries in the sub-menu, including "Create Array", "Create Cell Definition", "Create Cell Instance", "Create Copy of Cell Definition" and "Create Reference Point". Each is described individually in a section below.

6.1 Create Cell Definition

This command creates a brand-new symbol (cell) definition. You are asked to type in a name for the new symbol, and that symbol is created and opened. The newly created symbol will contain nothing but an x- and y- origin, and will be presented in all YALE's graphical viewports with exactly the magnifications that were there previously. If this command is accidentally selected, and it is not desired to create a new cell definition, simply type a <CR> instead of a cell name. This aborts the command as if nothing happened.

6.2 Create Cell Instance

This command inserts an instance of a previously defined cell within another. After selecting this command, you next show which symbol is to be inserted (via the most-recently-used-cell pop up). If you wish to place the symbol exactly as defined (no rotation or reflection), simply identify with MB3 exactly where you wish the origin to go. If the newly-inserted instance is to be placed with a rotation or reflection, again show where it is to go, but do so with MB1. You will then be presented with a pop up menu including such things as "flip lr", "flip ud", "rotate 3", "rotate 6", and "rotate 9". "flip lr" and "flip ud" refer to mirrorings (through the origin) left-right and up-down, respectively. The numbers in the rotate command are described in terms of a standard clock face. Imagine that in your original symbol, the hour hand points straight up. "rotate 3" means to rotate that hour hand until it points to the "3", and so on. Any combination of these transformations can be specified, and they are done one after another to the symbol before it is placed. In practice, at most 2 are required. For example, a "rotate 3" followed by a "rotate 6" would be exactly equivalent to a "rotate 9" selection, but there is no easy way to specify the the combination of a "rotate 3" followed by a "flip lr" command.

When you are finished specifying the transform, choose the bottom selection from the menu. The symbol will then be placed where the original MB1 was pressed.

As was the case with rectangles, the symbol instance is selected as it is inserted. In this way, if it is placed incorrectly, it is easy to issue a "Move Selected Objects" command and adjust it to the correct position.

The new symbol instance's origin (the intersection of the x- and y- origin reference points) is placed relative to the currently selected reference points in the currently open cell. An attempt to insert a symbol when no cell is open results in an error message in the tty viewport.

One often wishes to repeat this command over and over -- inserting a series of symbol definitions. Therefore, the "Create Cell Instance" command is repeatable. As soon as it is used, the "Rpt Cmd:" entry in the status viewport is updated to show that the currently repeatable command is "Create Instance". To repeat the command, simply press MB1 and MB2 at the same time. You will immediately be presented with the pop up menu of recently touched symbol definitions, and from then on, the command proceeds in exactly the same way as if you had gotten there by the usual path of getting the main YALE pop up, selecting "Create", and finally selecting the sub-menu entry "Cell Instance".

The "Create Cell Instance" command remains the repeatable command until another repeatable command is issued. There are only two others -- "Delete Selections", and "Add Selections".

6.3 Create Copy of Cell Definition

General SILT allows cells to be called over and over again with different parameters. YALE will eventually allow this, but now, although each cell definition may allow for stretchability, it can only be called with one set of parameters. If you wish to use a cell definition in two places with different parameters, you must make a copy of the cell definition, with a different name, and use that new definition with the new parameter set.

To make a copy of a cell definition, use the "Create Copy of Cell Definition" command. You will be asked to identify first the name of the cell definition to copy using the menu of recently touched symbol names. Next, you must type in the new name for the copy. The command can be aborted in two ways. You can bug outside the menu of most recently used cells, or you can type a <CR> when a typein is requested.

After the copy is made, YALE assumes that you are going to want to do something with it, so the previously open cell is closed, and the newly-created cell is opened and displayed in YALE's graphics viewports.

Page 22 blank

This command makes use of a scratch file on the remote host, so you must have a valid user name and password to use it. A scratch file called "_yale_scratch" is created there, and is only used during the execution of this command. The file can be deleted at any time, although it will be small in general, and can safely be left there.

6.4 Create Array

This command places a rectangular array of instances of the same symbol. At present, this command is a bit clumsy to use, but it is easier than putting in the symbols one at a time. When the command is issued, you are prompted in the tty viewport for the number of symbols to be placed in the x- and y-directions, and then for the x- and y- displacements. After this information is provided, you specify the position of the origin of the symbol in the lower left hand corner of the array. This is done in exactly the same way as it is for the simple "Insert Symbol" command, using MB3 to insert the members of the array in their standard orientation, and MB1 to specify a mirroring or rotation transformation, or some combination of the two.

When an array of symbols is inserted, all are selected, so that the array can be moved or deleted as a block if some error in spacing or placement was made.

If such an error is made, it is a good idea to correct it immediately, since The inserted array is not considered internally by YALE to be an array, but rather a series of individual symbol calls. Thus individual symbols in the array can be moved or deleted after the array is put in.

6.5 Create Reference Point

This command makes a new reference point. Reference points have a tree-like dependence, where the x- and y-origins serve as the the roots of the two reference point trees. When a reference point is moved, not only is all the geometry associated with it moved rigidly, but also all reference points that are relative to it. Thus, when a new reference point is inserted, it is placed relative to another reference point (which may be an origin).

After this command is issued, you are first prompted for the "parent" reference point (which is identified by bugging it with a mouse click). After this, its screen position is identified in the same way, and finally, the textual name for the new reference point must be typed in.

7. Setting Defaults

YALE has a few default values that can be set by the user that affect the actions of some of the editing and viewing commands. One of these, the default layer selected, has been described earlier. Every time an "Add rectangle" command is issued, the rectangle is made of material from the selected layer To select a layer, simply point to the appropriate stipple sample in the status viewport, and bug it with a mouse click. The other two default settings are new, and are described in the sections that follow. All are accessed initially by bugging the "Defaults" entry in the main YALE menu.

7.1 Setting Default Rectangle Widths

Each of the layers (metal, polysilicon, etc.) has a default width associated with it. In other words, when the default layer is set to be metal, and you insert a rectangle, the rectangle will be in metal, its length will be determined by the placement of the mouse clicks, and its width will be determined by the default width for the metal layer.

The current default widths for the various layers can be determined by looking at the number printed under the stipple samples in the status window. They can be changed using this command.

The "Set Default Widths" command is a multi-level pop up menu command. After bugging "Defaults" in the main menu and the "Line Widths" entry in the sub-menu, another sub-sub-menu of layers is presented. When one of these is selected, a fourth menu comes up that has as entries a set of typical small numbers and a "type in" option. If the width you want is in this menu, bug it, and you are done. If you need an unusual default width, bug the "type in" menu entry, and type in the number you want, followed by a <CR>. The change takes place immediately, and the information in the status window will be updated.

7.2 Setting Default Expansion Depths

In a complicated layout, when editing the top level symbol, you may not care about the internal details of the sub-cells. Filling in all the wires and transistors may even tend to clutter the screen toc much. Therefore, a method is provided to control the expansion depth when viewing a symbol.

The cell that is currently open is at level zero, as are the rectangles contained within it. The subcells of the open cell are, together with their included rectangles, at level 1. The cells and their rectangles contained within these sub-cells are at level 2, and so on.

Thus, if the expansion depth is set to zero, only the rectangles in the currently open cell are visible. Any sub-cells are simply indicated by an outline on the screen enclosing their name. All details of their interiors are hidden. If the expansion depth is set to 1, the rectangles within these cells are visible, but not the interiors of their sub-cells, and so on.

When YALE starts up, the expansion depth is set to a very large number (32767), so that essentially everything is visible.

Changing the expansion depth is much like setting the default rectangle widths, described above. After bugging "Default", and "Expansion Depth", you will be presented with a menu containing a set of small numbers, together with an entry marked "all", as well as the usual "typein" entry. The "all" entry sets the expansion depth to 32767, and the "typein" entry works just as it did in the "Set Default Widths" command -- just type in the required depth, followed by a <CR>. This command takes effect immediately, and the screen is redrawn with the new viewing parameters.

The default expansion depth affects all the YALE graphics viewports. Someday this should be changed to work on a viewport by viewport basis.

8. Select, Delete, and Move Commands

8.1 Selection

To move or delete objects in a symbol definition, it is necessary first to identify what those objects are. This is the purpose of the selection commands. The most commonly used selection command has already been described, and it selects a new reference point, symbol instance, or set of rectangle edges. If the selection is a symbol instance or set of rectangle edges, everything previously selected is first de-selected. If the selected object is a reference point the old selected reference point (with the same orientation -- vertical or horizontal) is first de-selected.

Add Selection. There can never be more than one reference point in each orientation (vertical or horizontal) selected, but it is often convenient to select many rectangles and symbol instances at the same time. This can be done with the "Add selections" entry of the sub-menu gotten from the "Selection" entry of the main YALE menu.

After bugging the "Add Selections" entry, the next mouse click is interpreted in the same way as MB2 is interpreted for a straight selection. The only difference is that previously selected items are not de-selected first.

Since it is common to use this command repeatedly to get many things selected at the same time, the command is repeatable in the same way as was "Create Cell Instance". If "Add Selections" is the current repeatable command, it is easy to use. Simply move the cursor to point to the object to be selected, and press MB1 and MB2 at the same time. The object pointed to will immediately be added to the selection list.

Select Ref. Pt. Dependents. The command to "Select Reference Point Dependents" selects all the rectangle edges and symbol instances that are dependent on some reference point. Before this happens, all other selected items are first de-selected. This command is most often used simply to find out what the reference point dependencies are -- that the items are selected is merely a side effect. The selection is a real one, however, and those items can be moved, deleted, and so on.

After bugging the entry for "Ref. Pt. Dependents", you are asked to bug a reference point, and it is items relative to this reference point that are selected.

8.2 Deletion

Delete Selections. There are three commands under the "Delete" entry of the main YALE pop up menu. The most important is "Delete Selections", which does just that -- all completely selected rectangles (those rectangles having all four edges selected) and selected symbol instances are deleted from the currently open cell. The deletion is permanent -- the deleted items cannot be retrieved from something like a "yank buffer" that is present in some textual and graphical editors.

The "Delete Selections" command is repeatable, and you can use it to delete a series of objects by alternately selecting an object with MB2, and then deleting the object by pressing MB1 and MB2 simultaneously.

Delete Cell Definition. The second delete command deletes an entire cell definition. This can only be done if that cell is called by no other cell in the layout. If this is not the case, an error message appears in the tty viewport.

Delete Reference Point. Finally, there is a command to delete a reference point. After bugging the "Reference Point" entry in the sub-menu, you will be asked to identify a reference point by bugging it. If the reference point has any geometry dependent on it, the deletion will not be carried out, and an error message will be presented in the tty viewport. If this is the case, remember that there is a command ("Select Reference Point Dependents", see 8.1) which will show all the dependent geometry.

8.3 Moving

Move Reference Point. The main reason for having reference points in YALE is to move them around to stretch and shrink cell definitions. This is done with the "Move Reference Point" command.

To do it, first bug the "Move" entry in the main menu, then the "Reference Point" entry in the sub-menu. Next, identify the reference point to be moved by bugging it, and identify its new location by bugging the screen. The reference point, together with all associated geometry (symbol instances and rectangle edges) should be moved rigidly along with it.

Move Selections. This command moves all the selected rectangle edges and cell instances rigidly to some other position in the cell. The motion references all items to the currently selected reference

Page 28 blank

point, but the dimensions of the moved objects all remain fixed, even if they were previously dependent on (placed relative to) two different reference points.

To use this command, bug the "Move" and "Selections" entry in the main YALE menu and in the sub-menu, respectively. You will then be asked to show how one point will move (remember that everything else moves the same way) by bugging its initial and then final positions. A common thing to do is to bug one corner of a selected rectangle where it is, and then to bug the position to which that corner should move. If all selected items are thought of as rigidly attached to the first point, the move command is equivalent to translating that point with everything attached to the second moused position.

Moving a collection of rectangle edges and cell instances does not de-select them. Thus, if an error is made, and they are moved to the wrong place, it is easy to give the move command again, and adjust the position again.

When only one of a rectangle's edges is selected, and the "Move Selections" command is issued, a somewhat surprising result can occur if the selected edge is moved so that it winds up on the other side of the unselected edge from which it was before. YALE simply remembers that, say, "the right edge is selected", and when what was the right edge becomes the new left edge, YALE still remembers that the right edge is selected. This situation does not often arise with normal cell transformations.

9. Input/Output

The SUN workstations do not have any local permanent storage, so all the data must be read from and written to remote files. This is done using the ethernet, and the files are available from any other ethernet host that provides remote file access, and supports the "leaf" protocol.

At the beginning of a YALE editing session, some initialization is done, including the opening of a connection to such a host, the specification of a user name, password, and of a main file name. After the initialization, all other input/output is done using the "Input/Output" entry in the main YALE pop up menu.

There are three kinds of remote files used by YALE. The first is the main input/output file that contains the current version of the layout being edited. Generally, this file is opened during initialization, is read then, and is written at the end of the YALE editing session. There is only one main input file used during a YALE session.

The second kind of file is the backup file. Whenever you give the "Backup" input/output command, this backup file is over-written with the current contents of the YALE memory. The previous backup version is lost. There is only one backup file used during a YALE session.

Finally, YALE supports (in a primitive way) library files. A library of cell definitions can be read in and combined with the user-defined cell definitions. Once they are combined, a copy is permanently kept in the user's layout file. Because of this, the best way to implement libraries is probably as small files, each containing only one or two library symbols. YALE has no restrictions on how many library files may be read in.

Of the three file types mentioned above, the library files are opened in a read-only mode, and the other two are opened in read-write mode. Errors may occur if the user you logged in as does not have the appropriate access to the files in question.

In general, the input/output structure could be improved a lot. There are, however, enough commands to do almost anything, albeit somewhat clumsily. Some of the defects can be remedied by going in with your favorite text editor and modifying the SILT files on the machine providing remote file storage.

9.1 Initialization

When you begin a YALE session, some initialization must be done. You will be asked for information about the main SILT file to be used for input and output. To find this out, YALE must connect to a host, log you in, and read the appropriate file. You will be prompted in the tty viewport (initially at the bottom of the screen) for a host (<CR> gives Shasta), a user name and password, and finally a file name. Type each of these followed by a <CR>. After this, you may issue any of the standard YALE commands.

If the file whose name you give in response to the request for the main file name does not exist, YALE assumes that you wish to create a new file by this name, and does so.

If you give a null file name (i. e. just type <CR>), YALE will not try to open a remote file. Thus, if you can experiment with YALE even if you do not have an account on a machine with an active leaf server. Just type <CR> in response to the user name and password requests. obviously, if you do this, you will have an empty layout, and will have to create everything from scratch. If you do this, and then decide that you really want to save your edits, you can always re-initialize the connection -- see the next paragraph.

The initialization commands are also available within an editing session as the menu entry "Initialize", found under the "Input/Output" entry in the main YALE pop up menu.

9.2 Close Connection

This command closes the currently open leaf connection.

9.3 Read Library

When this command is issued, you are asked for the name of a SILT library file, and that file is opened, and all the cell definitions contained therein are added to those in your layout.

9.4 User Name

The "User Name" command sends the remote system a new user name/password combination. At any point, YALE only keeps track of one such combination, so if you use this command to read in a library file, you had better use it again to connect back to the main directory if you wish to write out your main file after editing.

Page 32 blank

9.5 Write

The "Write" command writes out the contents of memory into the main file. If you have not yet specified it, you will be asked for the name of the main symbol to be used. The symbol name you select is added to the end of the SILT file produced so that it is the symbol expanded to create the whole layout. For example, if you select "foo" as your main symbol, a line of the form:

```
place foo() at (0, 0);
```

is put in as the only symbol call in the main begin-end block of the SILT file.

The reason that the main symbol name is requested during each YALE session instead of inferring it from the input file is that in this way it is possible to change it. YALE should be changed so that it is usually inferred, but can be changed with a specific "Default" or "Input/Output" command.

9.6 Write Backup

The "Write Backup" command is exactly like the "Write" command above, except that the information is written onto a backup file instead of the main YALE output file. The first time this command is issued, you will be asked for the name of the output file, and if you have not yet specified it, the name of the master symbol. After this information has been provided once, you will never be asked for it again during that session.

10. Miscellaneous YALE commands

Although the chapter is entitled "Miscellaneous YALE commands", this does not mean that they are not important. In fact, the commands documented here tend to be extremely important. All are accessed via only a single level of pop up menu.

10.1 Expand Cell

At any point in a YALE editing session, exactly one cell is open for editing. The "Expand Cell" command opens this cell. You show which cell is to be opened using the most-recently-used pop up menu mechanism. When a cell name is selected or typed in, the old cell is closed, and the new cell is presented in all the YALE graphics viewports. Everything in the cell (except, of course, for the vertical and horizontal origin reference points) is de-selected.

10.2 Re-reference

As each rectangle or symbol instance is added to an open cell, it is placed relative to the currently selected horizontal and vertical reference points. For rectangles, both edges are so placed, so without this command, there would be no way to make stretchable rectangles. The other common use for this command is when it is desired to make a cell flexible in some position where it was previously rigid. For example, suppose that it is desired to make a cell that was originally totally rigid so that it is stretchable in the x-direction. To do this, simply add a vertical reference point, and re-reference the items generally to the right to this new reference point. Move the new reference point around a little to see if all the stretching is done correctly, and then do some more re-referencing to correct errors.

The "Re-reference" command deals with the currently selected items, and leaves them in place, but re-references them to a new reference point that is identified by bugging it. It is effectively exactly the same as selecting the new reference point, and then "moving" all selected items by a distance of zero.

10.3 Window Manager

This command is available to allow an easy transition to the window manager from within YALE without requiring the user to take his hands of the mouse. It is exactly equivalent to typing the "brain escape character".

10.4 Show Cell Definition Names

This command displays a complete list of all the cell definitions currently in the layout. The names of the cell definitions are printed in the tty viewport, six to a line. The initial size of the tty viewport is small, and if there are many names, or if there are long names, they will not all fit in the viewport. Remember, however, that the tty viewport is like any other viewport on the display, and its size and shape can be modified with commands to the window manager. Just because the lines of text are not visible does not mean that they are not there.

10.5 Quit

The "Quit" command exits from YALE. It closes all open files on the remote machine, and closes the connection. The user is then returned to the SUN monitor. No warning is given if the edits have not been saved. In almost all cases, the next-to-last command is to write out the main file (see the next chapter).

The "Quit" command *must* be confirmed with a "y<CR>" (or a "Y<CR>"). To abort the "Quit" command, simply type anything else -- a raw <CR>, "n<CR>", or anything else. The editing session will then continue as if nothing has happened.

11. Errors

11.1 The '%' Command

If YALE (or any other user process running with the window manager) gets totally hung for some reason, there is an emergency abort that almost always works. Simply type the percent character, and everything should halt, returning you to the SUN monitor. This is a last-ditch attempt, and everything about the editing session is lost (except, of course, for any backup files that were written).

It works because there is a separate process watching the keyboard input, and even if some other process is stuck in a loop, the keyboard process still gets run regularly.

Use this command instead of the usual break command, since this one shuts up the mouse. (On some of the SUN monitors, the mouse is turned on by sending a command to the keyboard, and it continues to transmit its coordinates until it is turned off. The break key does not turn off the mouse, and the terminal is left in a useless state until the keyboard is unplugged or the terminal is powered down.)

11.2 SILT Parsing Errors

If all your editing is done using YALE, you should never have any trouble with this. On the other hand, if you go in with an editor and edit the SILT files produced by YALE by hand, it is not too hard to introduce syntax errors. YALE tries to read the file containing errors as best it can, and to continue after the error(s), but it will obviously sometimes be unsuccessful. There are many messages related to SILT syntax errors, and they should be self-explanatory. All such errors will be written in the tty viewport.

If such errors occur, the best bet is to go back to the original files and edit them until the errors go away.

11.3 Running out of space

No matter how much storage is available, it is possible to make a layout that is too big for it. Since YALE can run out of storage at totally unpredictable times, it is impossible to predict what the consequences are. Some of the storage is not efficiently reclaimed, and thus after editing for a long time, you may run out of space even though the layout has not gotten much more complicated. Writing out the file, restarting YALE, and reading in the file again can sometimes help.

Some mechanism should be created in YALE to cache data on the remote host, but this has not yet been done.

If the layout is just too big to deal with, YALE can be used to put together parts of it, and all the parts can be combined by hand using SILT to wire together the pieces. The SILT translator, running on the VAX, can handle *very* large layouts.

A warning about low free storage is hard to give, since a single YALE command can use up a large amount of it. For example, creating a large array of symbol instances can chew up free storage in a hurry. When free storage is exhausted, the message "Out of free storage" is put up in the tty viewport, and YALE attempts to leave its data structures in a reasonable state, but this is not always possible. The best advice is to back up your work frequently when you think that you may be low on free storage.

11.4 Other Size Limitations

YALE has no absolute cell size -- a symbol can call as many rectangles as it wants. There is an implementation restriction on the number of symbol calls that a given symbol can make. A single cell must contain less than 300 symbol calls, but it is not as simple as that. If symbols are deeply nested, this number can be reduced. If, during an "Expand Symbol" command, the error "Too many symbol calls" occurs, try reducing the number of symbol calls in symbols with a large number of them.

Another scarce resource is something called "Sun Instance Numbers". The number of these is a compiled-in constant, so can be changed by re-compiling. Each cell definition, cell instance, reference point, and rectangle has a Sun Instance Number. When an object is deleted, its Sun Instance Number is returned to a pool of free numbers. Sun Instance Numbers are used to look up quickly information about items displayed on the screen.

11.5 Bugs

Some of the error messages are prefixed by the word "bug". These should never be printed out. If one is, please report it to the YALE maintainer, together with as much information about what was going on as possible.

Reports of other bugs are also welcome.

I. Window Manager Command Outline

Page 38 blank

This is a list of all the commands available in the window manager. Each command is followed by the page number on which it is more fully described.

Zoom In (MB1) 8
Zoom Out (MB2) 8

Brain Escape Character 10
Center Window 9
Create Viewport 9
Delete Viewport 9
Move Edge 9
Move Viewport 9
Redraw 10
Return to Yale 10
Toggle Grid 10

% (emergency abort command) 36

II. YALE Editor Commands

This is a list of all the commands available in the YALE layout editor. Each command is followed by the page number on which it is more fully described. Those commands followed by an asterisk (*) are repeatable.

```
Add Rectangle (MB1) 17
Select Item (MB2) 17

Create Array (of instances) 21
Create Cell Instance (*) 19
Create Cell Definition 19
Create Copy of Cell Definition 20
Create Reference Point 21

Default: Set Default Width 23
Default Expansion Depth 23

Delete Cell Definition 26
Delete Reference Point 26
Delete Selections (*) 26

Expand Cell 33

Input/Output: Initialize 30
Input/Output: Close Connection 30
Input/Output: Read Library 30
Input/Output: User Name 30
Input/Output: Write 31
Input/Output: Write Backup 31

Move Reference Point 26
Move Selections 26

Re-reference Selections 33

Select: Add (to) Selections (*) 25
Select Reference Point Dependents 25

Show Cell Definition Names 34

Quit 34

Window Manager 33
```

References

1. Brown, David J., and Nowicki, William I. A Package of Graphics Primitives for SUN. Stanford internal documentation of the Raster Operation routines.

2. Cheriton, David R., and Mann, Timothy P. The V Kernel: A Distributed Message-based Kernel. Stanford, 1982.

3. Davis, Tom, and Clark, Jim. SILT: A VLSI Design Language. Stanford, 1982.

4. Mogul, Jeffrey. Leaf and Sequin Protocols. Unpublished paper, Stanford University, March 12, 1981

NOTES ON

GATE ARRAY WIRING AREA PREDICTION

via Statistical Modelling

VLSI file #102282

Abbas El Gamal

Information Systems Lab
Stanford University

presented at the D.A. Workshop, Burlington VT,
September 1982

Gate Array Wiring Area Prediction Via Statistical Modelling

Abbas El Gamal

Information Systems Lab

Stanford Univ., Ca.

Why is the prediction needed?

I. Deciding the size and organization of the arrays

II. Deciding whether a given logic circuit can be implemented on a specific gate array.

- Measure for difficulty of routing.

How should the prediction be?
done ?

I* A large sample of representative circuits are tried on different array sizes and organizations (this includes past experience etc.)

II * Given the circuit, do the placement and the routing to find out if circuit fits and if routing is difficult.

Can prediction be done more economically ?

Prediction Via Statistical Modelling

Previous Work:

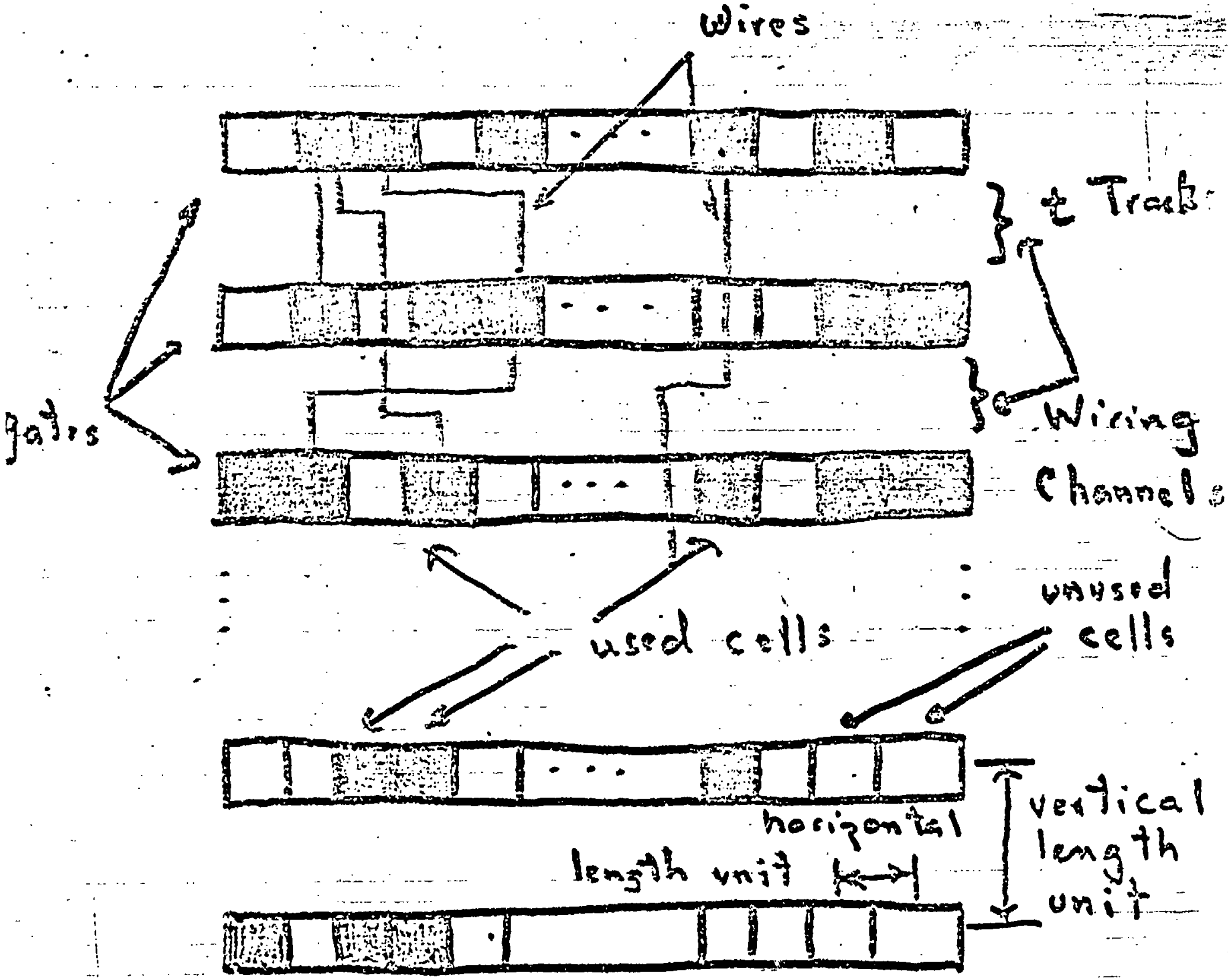
1. I. Sutherland and D. Oestreicher, "How big should a printed circuit board be?" IEEE Trans. Comp., May '73.

2. W. Heller, W. Mikhail and W. Donath, "Prediction of wiring space requirements for LSI," Design Automation & Fault-Tolerant Computing, pp. 117-144, 1978.

3. A. El Gamal, "Two-Dimensional Stochastic Model for Interconnection in Master Slice Integrated Circuits", IEEE Trans. Circuits and Systems, Feb., 1981.

4. A. El Gamal and Z. Syed, "A Stochastic Model for Intercon. in Custom Integrated Circuits", IEEE Trans. Cts. and Systems, Sept., 1981.

Gate Array Organization For new Mod



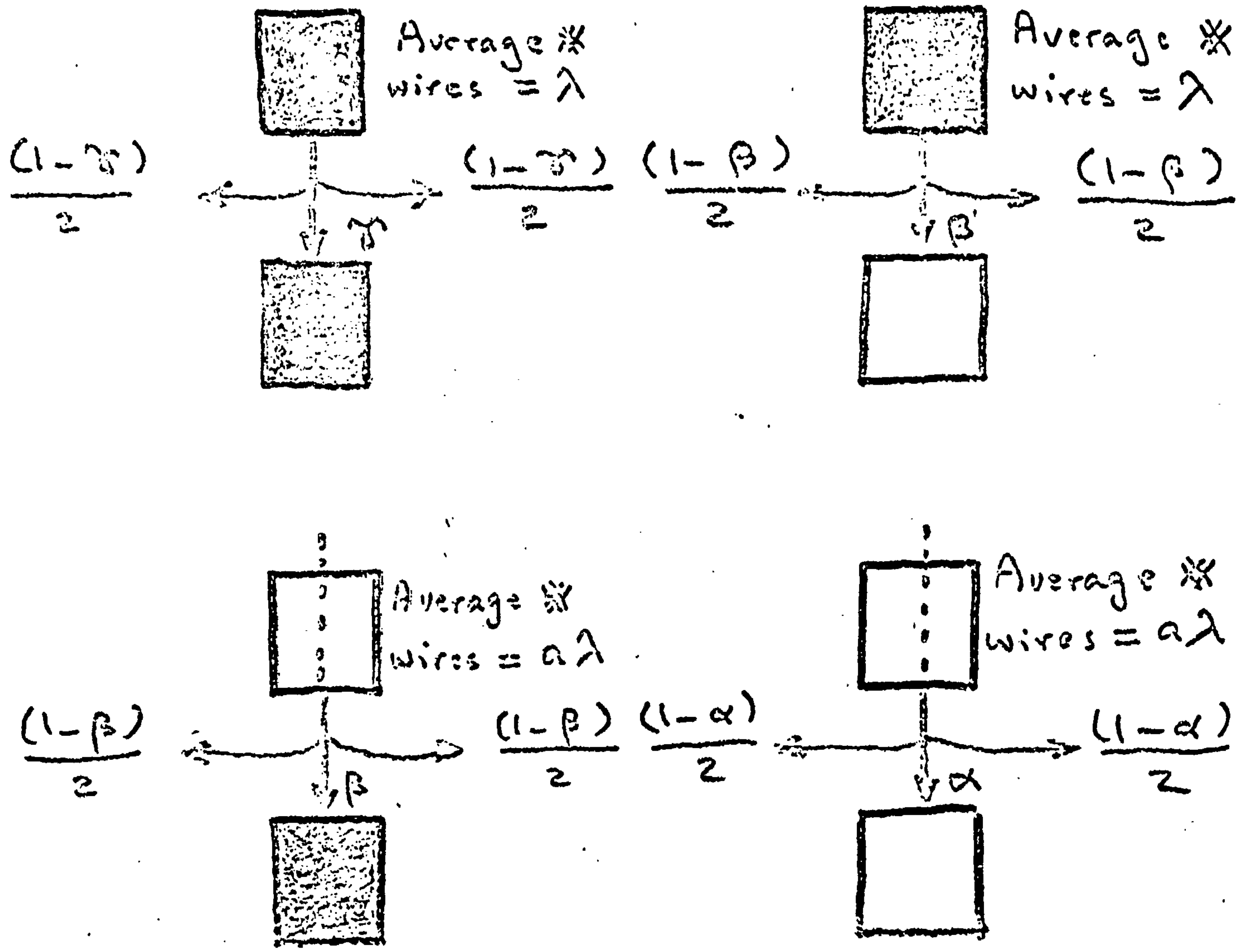
# of gates ≙ N

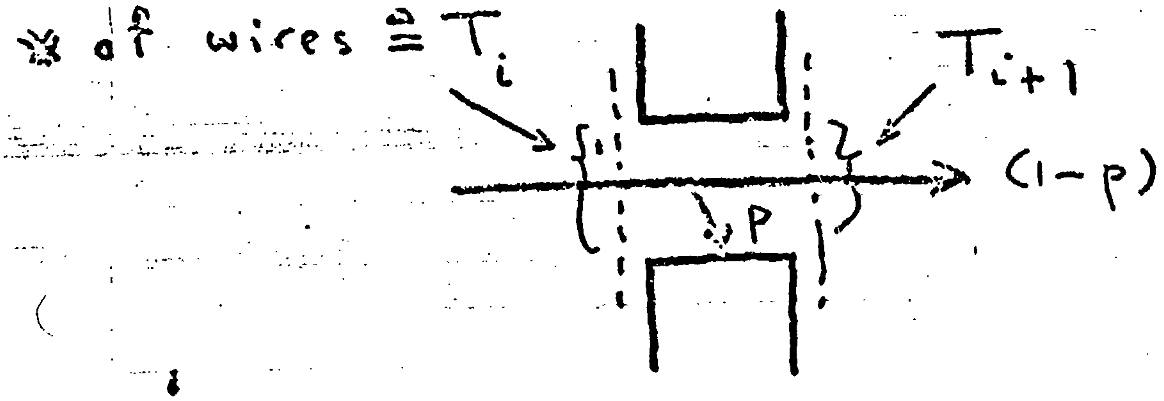
F ction of used cells ≙ u

$\overline{R}_H$ ≙ horizontal average wire length

$\overline{R}_V$ ≙ vertical " " "

Wire Trajectories Model





Results : Parameters given : $\lambda, \alpha, \beta, \gamma, p, u, N$

- $\bar{R}_V = \dfrac{\bar{u}(u\bar{\gamma} + \beta + \bar{\beta}\bar{u})}{1 - \bar{u}((\alpha + \bar{\alpha}\bar{u}) + u\bar{\beta})}$

- $\bar{R}_H = \dfrac{1}{p}\left[(u\bar{\gamma} + \bar{u}\bar{\beta}) + \bar{R}_V \cdot (u\bar{\beta} + \bar{u}\bar{\alpha})\right]$

- $Q = \dfrac{u(\beta + \bar{u}\bar{\beta} + u\bar{\gamma})}{1 - \bar{u}(1 + u(\alpha - \beta))}$

- $ET_i = \dfrac{\lambda}{p}\left[u(u\bar{\gamma} + \bar{u}\bar{\beta}) + \bar{u}a(\bar{u}\bar{\beta} + u\bar{\alpha})\right]$

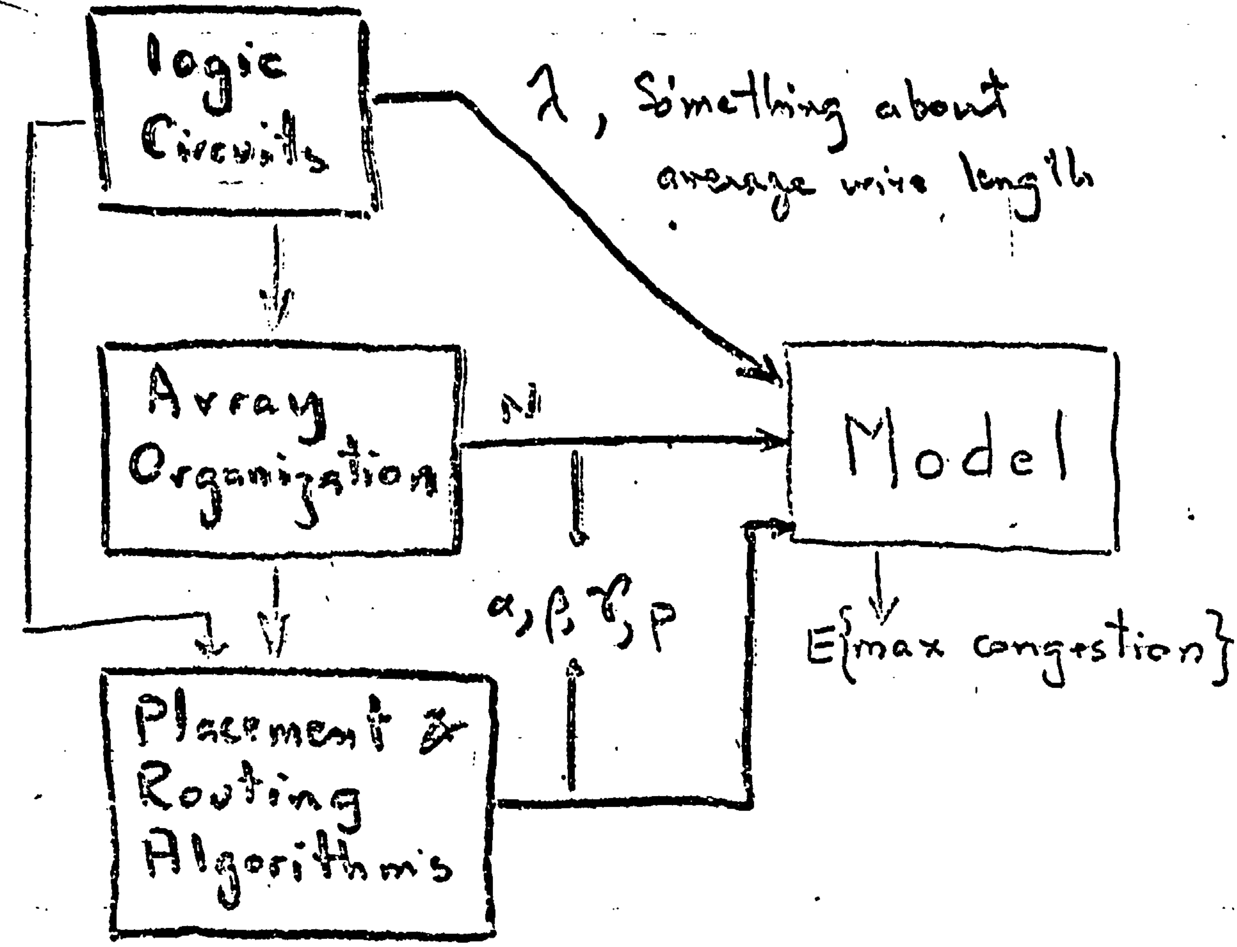
- Developed an upper bound to $E \max_i T_i$, i.e. maximum congestion

- Can compute wire length distribution.

Two Important Questions

- How can parameters be estimated?
- Given "correct" parameters, how good is the prediction?

Parameter Estimation


logic Circuits
Array Organization
Placement & Routing Algorithms
Model
λ, Something about average wire length
N
α, β, γ, p
E{max congestion}

u (utilization) is a parameter that can be varied within certain reasonable limits

Example: (real chip)

Gate Array Parameters:

number of rows = 22

number of columns = 191

number of tracks = 16

Measured parameters of chip:

$ET_i = 9.82$ $\qquad \alpha = 0.775$

$\beta = 0.39$ $\qquad \gamma = 0.166$

$\lambda = 1.465$ $\qquad \bar{R}_H = 13.5$

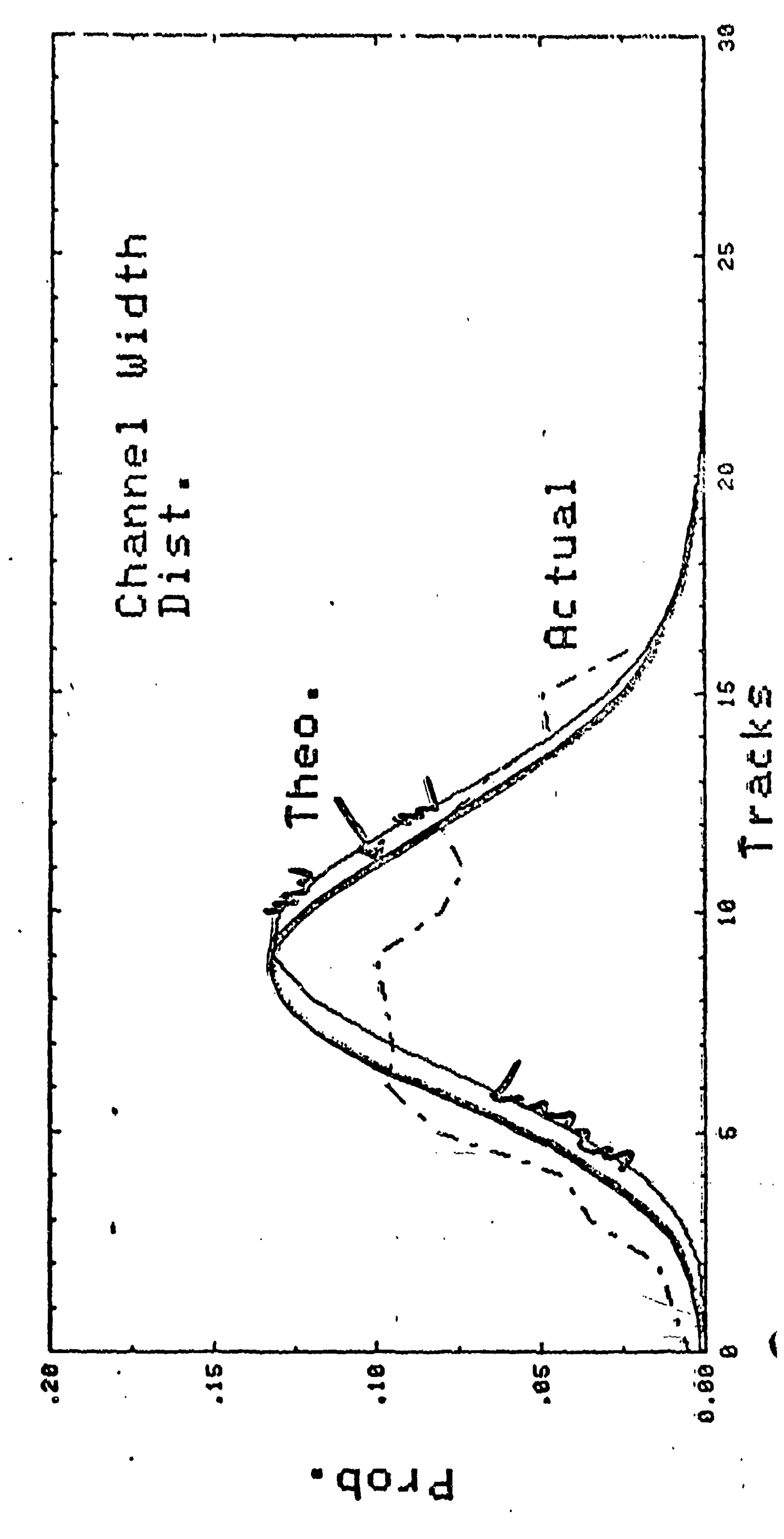
$\bar{R}_V = 1.46$

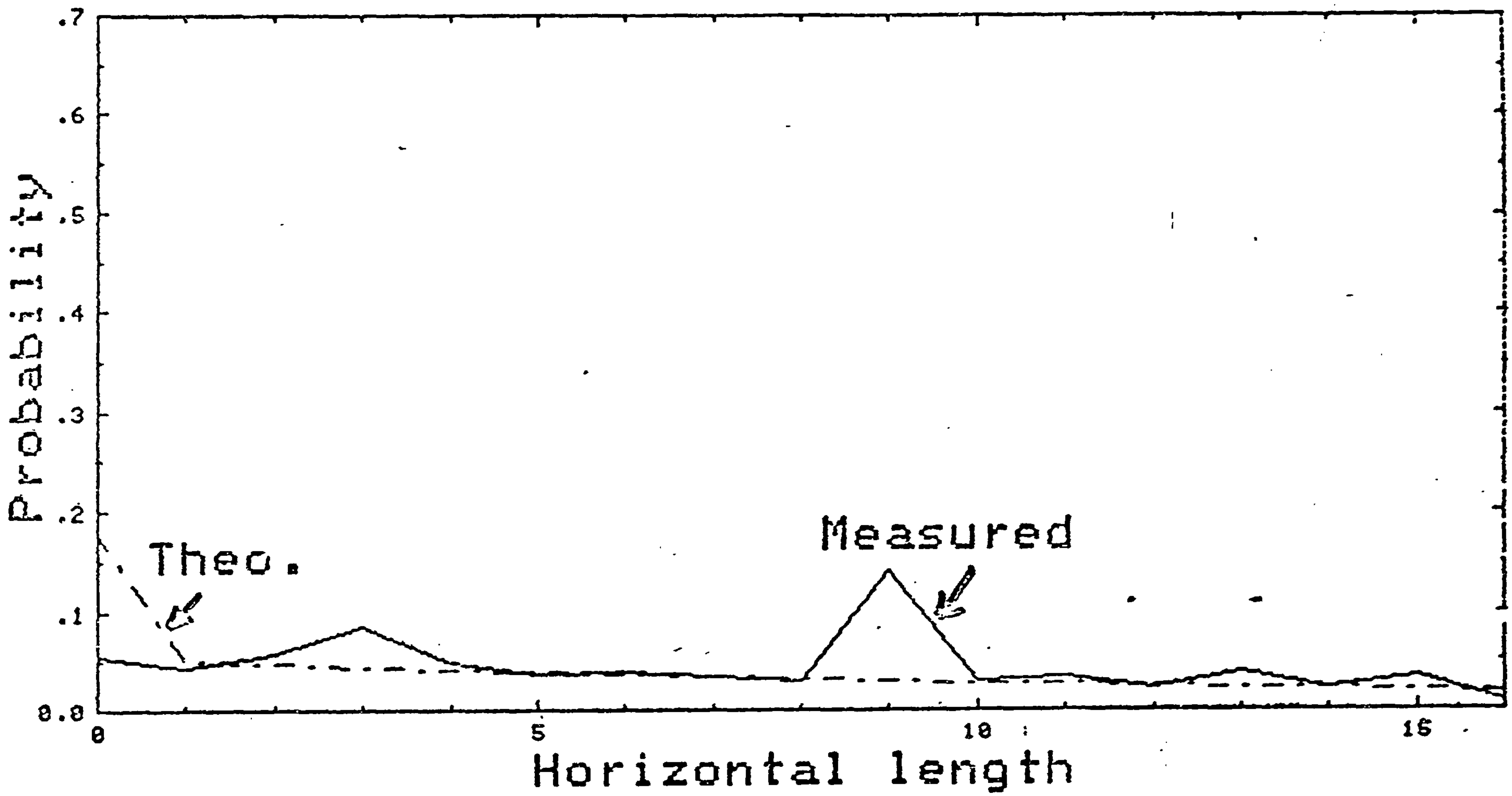
Computed parameters

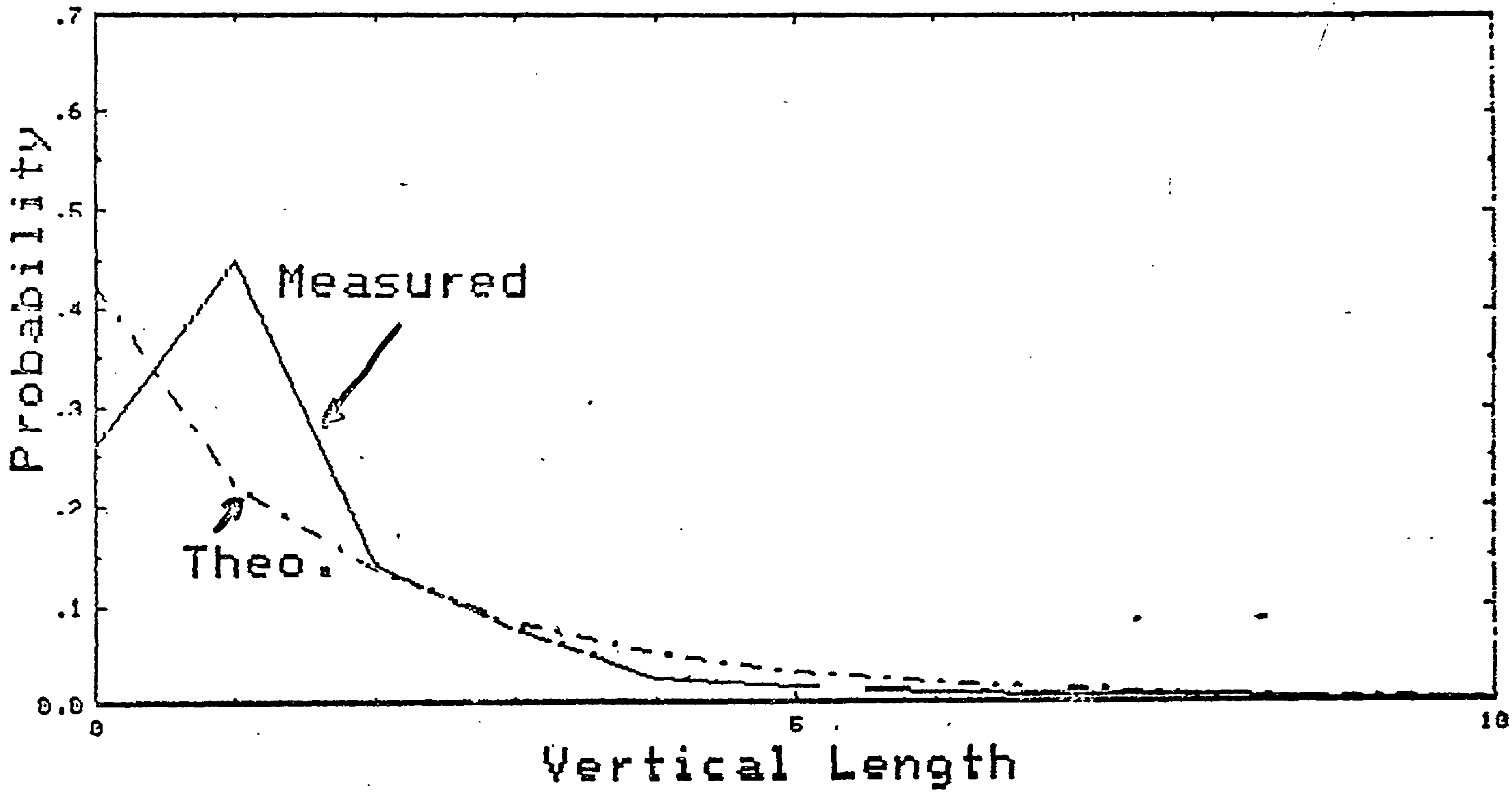
$\rho = 0.099$ $\qquad ET_i = 9.9$

$a = 1.4$

Expected max. Congestion ≤ 21.8


Channel Width
Dist.
Theo.
Actual
Prob.
.20
.15
.10
.05
0.00
0
5
10
15
20
25
30
Tracks


Probability
.7
.6
.5
.4
.3
.2
.1
0.0
0
5
10
15
Horizontal length
Theo.
Measured


Probability
.7
.6
.5
.4
.3
.2
.1
0.0
0
5
10
Vertical Length
Measured
Theo.

Application of Model:

- Is it better to use small u, number of tracks
  Or high n, large ″ ″ ″ ?

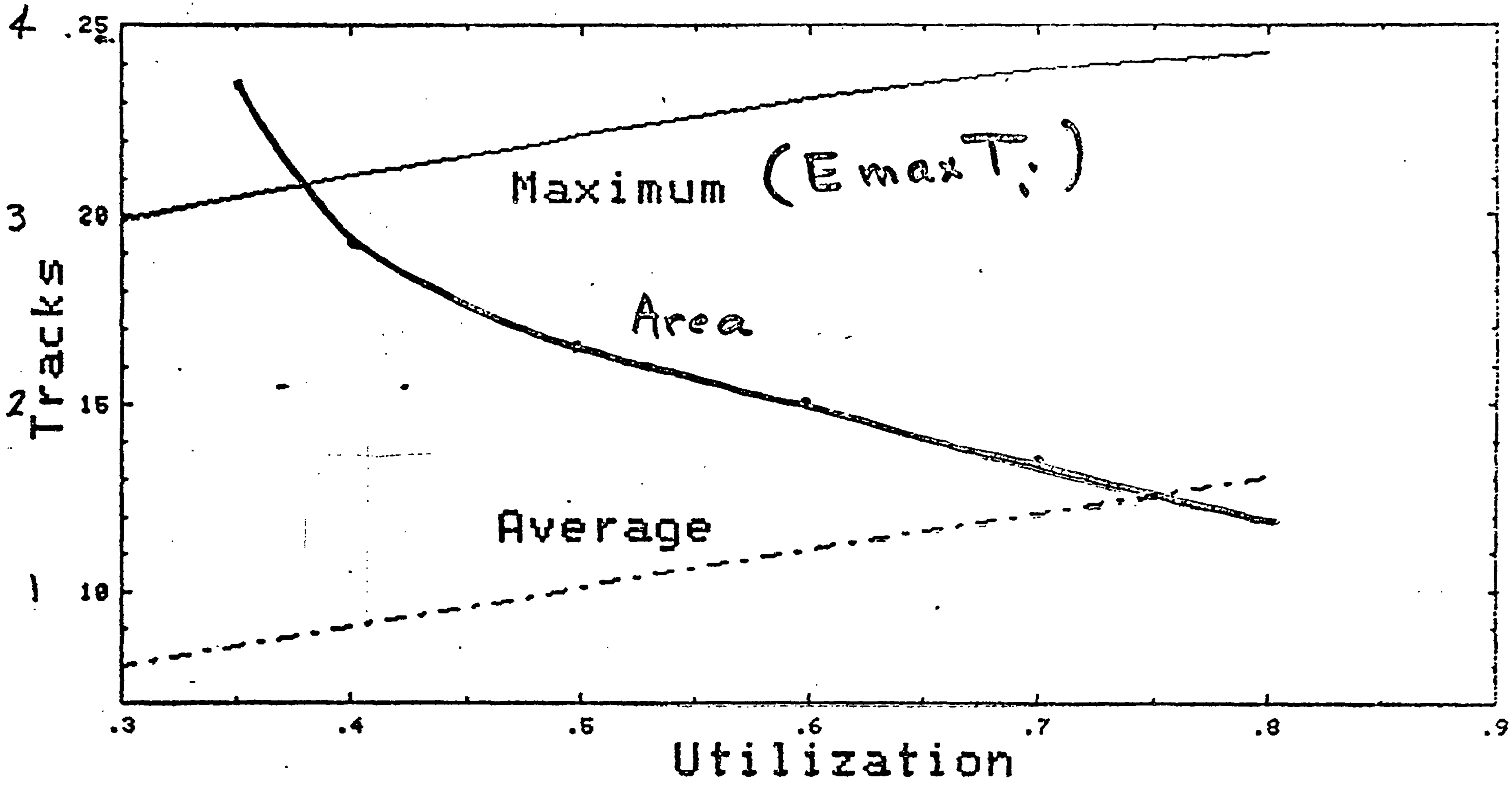
Experimental data (same example)

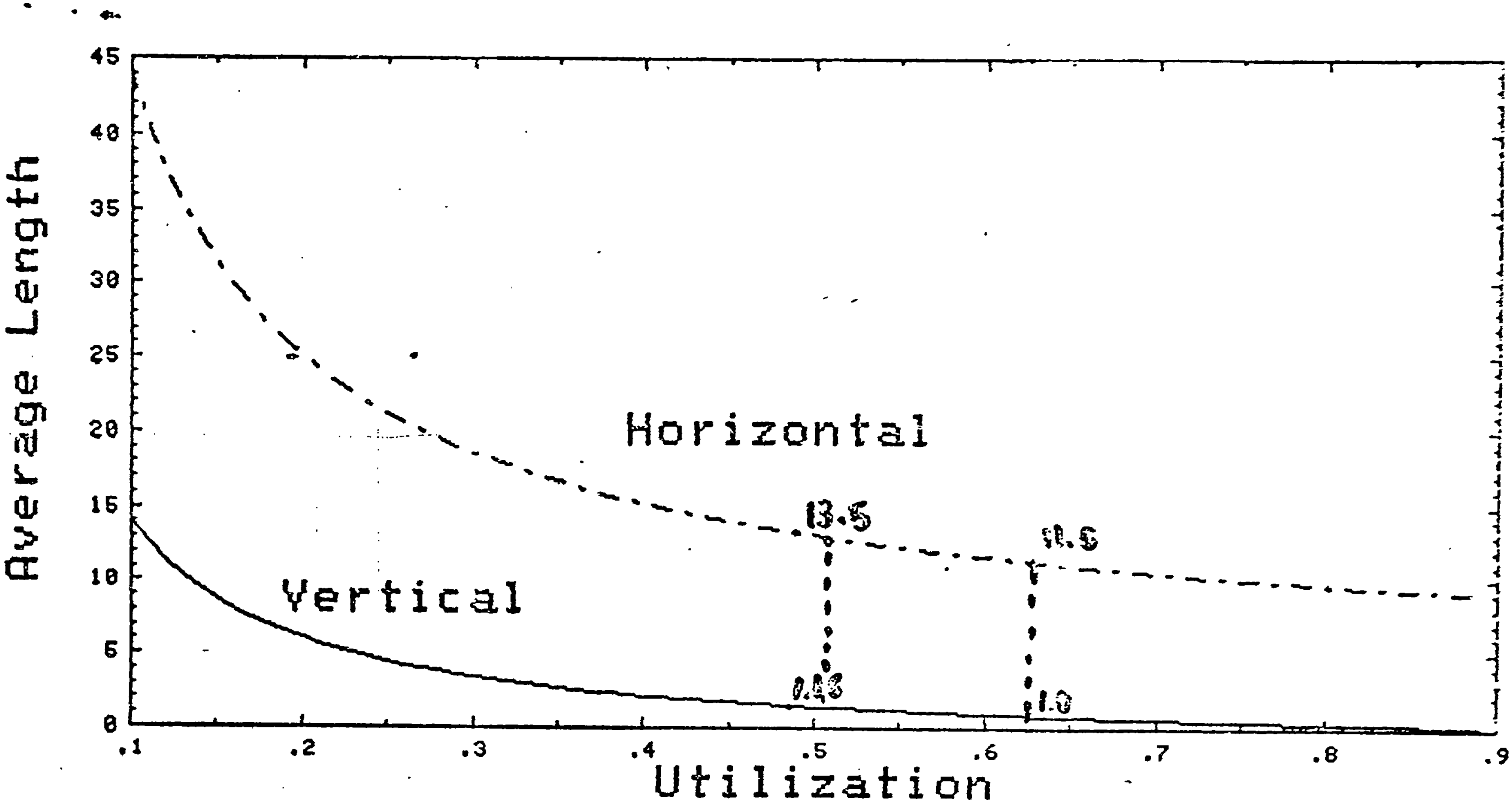
# Gates	# Tracks	% Auto-Wiring
3200	16	71.6 %
3200	19	78.7 %
3200	24	96.2 %
4200	16	93.3 %
4200	19	97.9 %
4200	24	99.3 %

$$\frac{\text{Area for 3200, 25 tracks}}{\text{″ ″ 4200, 19 tracks}} = \frac{1}{1.18}$$

Wire length (not known)

From Model;


4
3
2
1
.25
20
15
10
Tracks
Maximum (E max T_i)
Area
Average
.3
.4
.5
.6
.7
.8
.9
Utilization


Average Length
45
40
35
30
25
20
15
10
5
0
.1
.2
.3
.4
.5
.6
.7
.8
.9
Utilization
Horizontal
Vertical
13.5
11.5
1.43
1.0

From model it is suggested that the same example can be routed with comparable % completion with 25 tracks and $U = 0.8$ (instead of 0.48)

, and with

$\bar{R}_H = 10$ (instead of 13.5)

$\bar{R}_V = 0.5$ (" " 1.46)

This results in area saving of ≈ 25% and decrease of average wire length of ≈ 35%.

The Compilation of Regular Expressions into Integrated Circuits

ROBERT W. FLOYD AND JEFFREY D. ULLMAN

Stanford University, Stanford, California

Abstract. The design of integrated circuits to implement arbitrary regular expressions is considered. In general, a regular expression with n operands may be converted into a nondeterministic finite automaton with at most n states and n transitions. Instead of converting the nondeterministic device to a deterministic one, two ways of implementing the nondeterministic device directly are proposed. One approach is to produce a PLA (programmable logic array) of approximate dimensions n rows and $2n$ columns by representing the states of the nondeterministic finite automaton directly by columns. This approach, while theoretically suboptimal, makes use of carefully developed technology and, because of the care with which PLA implementation has been done, may be the preferred technique in many real situations. Another approach is to use the hierarchical structure of the automaton produced from the regular expression by the McNaughton-Yamada algorithm to guide a hierarchical layout of the circuit. This method produces a circuit $O(\sqrt{n})$ on a side and is, to within a constant factor, the best that can be done in general.

Categories and Subject Descriptors: B.1.2 [**Control Structures and Microprogramming**]: Control Structure Performance Analysis and Design Aids—*automatic synthesis, formal models*; B.7.2 [**Integrated Circuits**]: Design Aids—*layout*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*routing and layout*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*classes defined by grammars or automata*

General Terms: Algorithms, Design, Languages, Theory

Additional Key Words and Phrases: Regular expression, nondeterministic finite automaton, programmable logic array, circuit area

1. *Introduction*

There are a number of projects, such as [4, 6, 13], whose goal is "silicon compilation," that is, the automatic layout of circuits from their behavioral description. These projects tend to be oriented around the design of computerlike circuits, certainly an important goal, but one that is analogous, in the software domain, to implementing languages suitable for writing operating systems, but little else. It appears that the "FORTRAN" of circuit implementation must be quite general-purpose, allowing us to specify a great variety of different kinds of circuits and to implement anything we can specify, with a fair degree of efficiency.

It is the purpose of this paper to discuss only one possible component of such a general-purpose language, a regular expression facility. Regular expressions are capable of specifying any finite-state process, although they are *not* always as succinct

This work was supported in part by DARPA Contract MDA903-80-C-0102 and in part by the National Science Foundation under Grant MCS 79-04528.

Authors' address: Computer Science Department, Stanford University, Stanford, CA 94305.

as other representations [3]. Fortunately, there is a common class of finite-state processes for which regular expressions appear very well suited indeed. In the software world, lexical analyzers, which recognize the tokens (e.g., identifiers, keywords) of a programming language, have been generated automatically from regular expressions defining the tokens. Regular expressions also make a good language for describing patterns to be matched by a text editor. Aho and Ullman [1] describe these and other software applications of regular expressions.

In the hardware world, regular expressions may be suited to describing certain controllerlike processes, where it is desired that we signal "events," where each "event" consists of a sequence of significant input signals, perhaps interspersed with arbitrary numbers of irrelevant signals. We shall later give a design example for a simple device of this sort. On the other hand, regular expressions are not very good for describing counting processes. For example, the event "876 zeros" is most naturally described by the regular expression $00\cdots 0$ (876 times). Obvious techniques for producing a circuit from this expression will only succeed in producing a unary counter with 876 distinct memory elements, rather than a binary counter with ten memory elements. Extensions to the regular expression language can alleviate this problem somewhat, but the fact remains that regular expressions cannot be billed as a panacea, even if we restrict our domain of interest to sequential processes. However, they do represent a promising approach to the automatic design of some components, and they probably have a place in any general-purpose compiled circuit design language.

2. *The Circuit Model*

To be specific, let us assume that circuits are implemented in the nMOS technology, using the Mead–Conway [11] design rules. However, what we say applies to any technology in which

(1) 2-input logical operations can be implemented in constant space;
(2) wires have a fixed constant width, and signals can be driven through the wire in an acceptably short time by a driver no larger than the wire itself;
(3) there is a particular number of wires, at least two, that may occupy the same area; the number 3 applies to the nMOS technology.

This model of integrated circuits is discussed in [2, 14], for example.

3. *Regular Expressions and Nondeterministic Automata*

We assume the reader is familiar with finite automata theory as discussed in [5], for example, and we only sketch the essential details here. Regular expressions are built from an alphabet Σ (in practice, Σ might be the set of ASCII characters, for example) using the following rules.

(1) For each a in Σ, a is a regular expression denoting $\{a\}$, that is, the set consisting of one string; that string is of one symbol, a.
(2) $\varnothing$ and ϵ are regular expressions denoting, respectively, the null set and $\{\epsilon\}$, that is, the set consisting of the *empty string* (zero-length string) only.
(3) If R_1 and R_2 are regular expressions denoting sets of strings S_1 and S_2, respectively, then $(R_1) + (R_2)$, $(R_1)(R_2)$, and $(R_1)^*$ denote $S_1 \cup S_2$, $S_1 S_2$, and S_1^*, respectively. Here $S_1 S_2$ is the *concatenation* of sets S_1 and S_2, that is,

$$\{xy \mid x \in S_1 \wedge y \in S_2\}.$$

Also, S_1^*, the *closure* of S_1, is

$$\{\epsilon\} \cup S_1 \cup S_1S_1 \cup S_1S_1S_1 \cup \cdots .$$

That is, $(R)^*$ means "zero or more occurrences of R."

(4) Parentheses may be dropped when they are implied by the following precedence order: closure highest, then concatenation, then union. For example, $a + bc^*$ is grouped $a + (b(c^*))$ and stands for the set of strings

$$\{a, b, bc, bcc, bccc, \ldots\}.$$

Sometimes it is useful to extend the regular expression language in several ways that do not affect the collection of sets of strings we can define. For example, LEX [8], the UNIX lexical analyzer generator, uses . to stand for "any character," that is, the expression $a_1 + a_2 + \cdots + a_n$, where the a_i's are all the symbols in Σ. Also, $(R)^+$ stands for the *positive closure* of R, that is, $R + RR + RRR + \cdots$, or "one or more occurrences of R." The expression $(R)?$ means "zero or one occurrence of R," that is, $\epsilon + R$.

A *nondeterministic finite automaton* (NFA) is conventionally represented by a directed graph, whose nodes are *states*, and an arc from state p to state q can be labeled by any symbol from Σ or by ϵ, the empty string. We allow multiple arcs between two states, but we usually represent these arcs by a single arc with more than one label. One state is designated the *start state*, and one or more states are designated *accepting* or *final* states. The NFA *accepts* a string $a_1a_2 \cdots a_n$ if there is a path from the start state to some accepting state, and the labels of the arcs along that path read $a_1a_2 \cdots a_n$. Note that ϵ may be a label of one or more of those arcs, but ϵ is "invisible," that is, it can appear any number of times along the path without appearing in the string accepted.

Example 1. Let us now take an example of how a sequential process can be represented by regular expressions and by an NFA. Consider a control unit that receives a sequence of two bits, which it interprets as a command according to the code

$$\begin{aligned} 00 &= \text{add}, \\ 01 &= \text{subtract}, \\ 10 &= \text{load}, \\ 11 &= \text{load complement}. \end{aligned}$$

For simplicity, we assume that the source of commands is "well behaved"; we never receive anything but two bits at consecutive times, nor can a second command be received while the previous command is being processed.

The output consists of three lines, A, C, and L, which, respectively, cause (A) add the memory buffer to some particular register, (C) complement the memory buffer, and (L) load the memory buffer into the register. When the C signal is sent, the controller waits for a completion input signal (X) before sending the A or L signal. As the machine is synchronous, we actually have a fourth input symbol besides 0, 1, and X in our alphabet Σ. We use N to indicate that no command bit or completion signal is present on the input.

As an aside, we note that the input alphabet $\Sigma = \{0, 1, X, N\}$ should be regarded as consisting of logical, rather than physical, inputs. For example, in practice there might be three binary input lines: "command bit," "command present indicator," and "completion." The 0 input is represented by a command bit of 0, with the

FIG. 1. Actual-to-logical input interpretation.

Command bit	0	0	0	0	1	1	1	1
Command present	0	0	1	1	0	0	1	1
Completion	0	1	0	1	0	1	0	1
Logical input	N	X	0	0	N	X	1	1

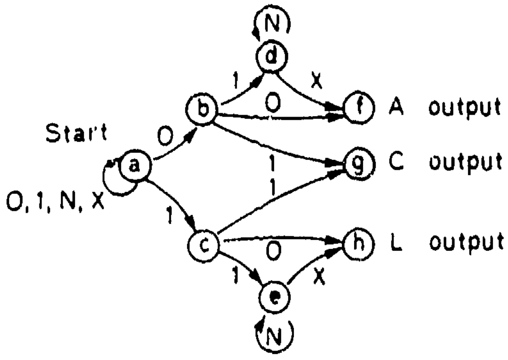


FIG. 2. An NFA fpr the controller example.

command-present bit set to 1. The completion bit can be ignored, as a 1 on that line while the command is present violates our assumption that commands do not overlap. The interpretation of the three bits as input symbols from Σ is shown in Figure 1.

The regular expression for the "add" output signal is given by

$$A = .^*0(0 + 1N^*X),$$

where . stands for "any input symbol." That is, we wish to signal an addition if after any sequence of inputs we see a 0 followed immediately by either

(1) another 0, completing the command 00 = add, or
(2) a 1, completing the command 01 = subtract, followed by any number of N's and an X. In this case, we assume the "complement buffer" signal C is sent after receiving 01. The N's represent "clock ticks" while we wait for the completion signal. When the X is received, we know the buffer has been complemented and immediately issue the "add" signal.

Similarly, we can specify the conditions under which we should emit the C and L signals by

$$C = .^*(0 + 1)1,$$
$$L = .^*1(0 + 1N^*X).$$

We shall subsequently discuss an algorithm to convert any regular expression to an NFA with some arcs labeled ϵ. However, we first illustrate the NFA concept with one NFA for the controller; this NFA, shown in Figure 2, uses no ϵ-arcs, but it does have nondeterminism, in the sense that it can be in more than one state at the same time.[1] For example, suppose we have input $N01$. We begin in state a, the start state. The only arc with label N leaving state a leads back to a. Thus, after the first input symbol we are only in state a. The next input, 0, labels arcs from a to a and b, so after the second input we are in those two states. Then we look for arcs out of a or b labeled 1, and we find them from a to a and c, and from b to d and g. Thus, after the third input we are in a, c, d, and g. Since g is a final state, we "accept" $N01$. In practice, state g represents the C signal, which is appropriate, since our input is one instance of the "subtract" command. □

[1] Nondeterminism should not disturb us here. The NFA is a mathematical abstraction, and its implementation in hardware is quite different from that of its deterministic counterpart (a DFA), which is guaranteed to be in only one state at a time.

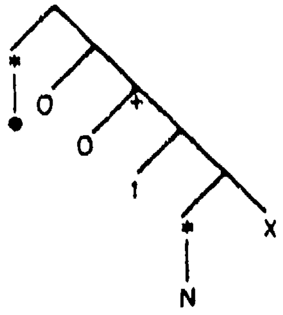


FIG. 3. Parse tree for .*0(0 + 1N^*X).

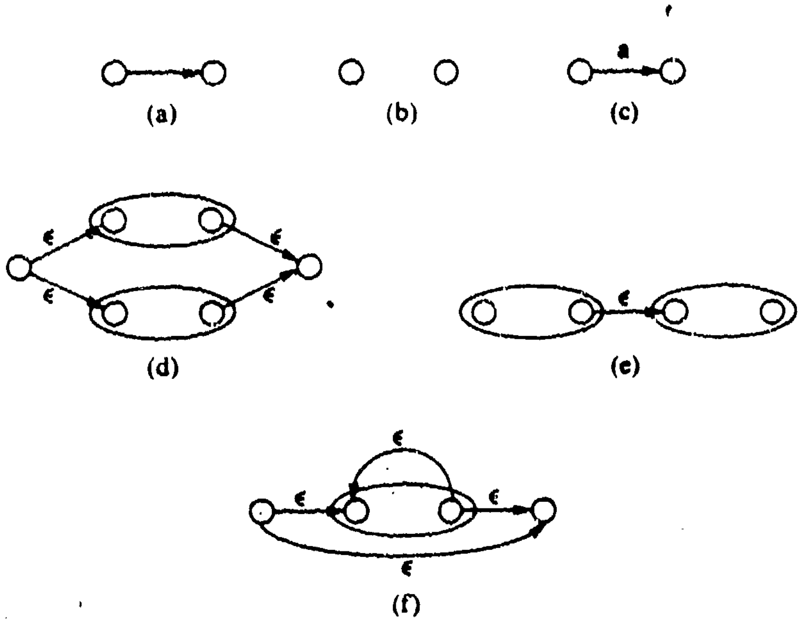


FIG. 4. The McNaughton–Yamada constructions for (a) ϵ, (b) $\varnothing$, (c) a in Σ, (d) union, (e) concatenation, and (f) closure.

4. *The McNaughton–Yamada Construction*

There is a well-known recursive algorithm, due to McNaughton and Yamada, for converting regular expressions to NFAs with some ϵ-arcs. The algorithm produces NFAs for the regular expressions $(R_1) + (R_2)$, $(R_1)(R_2)$, and $(R_1)^*$, given NFAs for R_1 and R_2. To begin, we must "parse" a regular expression. That is, we view the regular expression as a *parse tree*, where leaves represent symbols in Σ (or ϵ or $\varnothing$ if needed), and interior nodes represent the application of union, concatenation, and closure operators to subexpressions. For example, the parse tree for expression A of Example 1 is shown in Figure 3. See [1] for a description of how parse trees for regular expressions can be constructed.

The McNaughton–Yamada algorithm [5, 10] constructs for any regular expression an NFA with one start state and one final state. It is conventional to draw NFAs with the start state on the left and the final state on the right. Figure 4a–c shows the basis of the construction, the two-state NFAs that recognize ϵ, $\varnothing$, and any particular a in Σ, respectively. Figure 4d–f shows how NFAs M_1 and M_2 for regular expressions R_1 and R_2 are combined to get NFAs for $(R_1) + (R_2)$, $(R_1)(R_2)$, and $(R_1)^*$. Simple modifications of construction (f) give us the positive closure ($^+$) and zero-or-one (?) operators. In the first case, eliminate the ϵ-arc from the new start state to the new final state, and in the second case, eliminate the backward arc.

Example 2. The NFA constructed from the expression $A = .^*0(0 + 1N^*X)$ is shown in Figure 5. There, and henceforth, we adopt the convention that final states

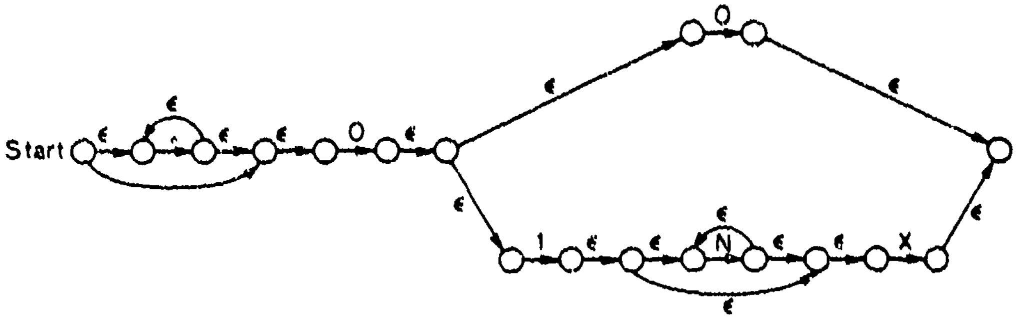
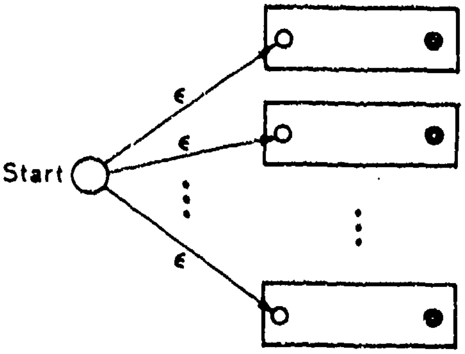


FIG. 5. NFA for expression *A*.

FIG. 6. Combining several NFAs into one.



are indicated by double circles. Note the great superfluity of ϵ-arcs. Many of these can be eliminated by considering special cases in the recursive construction rules. They can all be eliminated by replacing an ϵ-arc from state p to state q by arcs from p to whatever states q's arcs go to, and then, if q is final, making p final. Then we may eliminate q if it is not the start state and it no longer has any entering arcs. See [1] for details. □

There is one more step to the construction of an NFA from a collection of regular expressions. We introduce a new start state with ϵ-arcs to the start states of the NFAs for all the regular expressions in the collection. This construction is illustrated in Figure 6. Note, however, that the various final states of the combined NFA are not indistinguishable. Each represents one of the output signals for the device. In a sense, the NFA of Figure 6 represents an extension to the usual concept of an NFA, since there are differing output signals associated with the different final states.

5. *A PLA Implementation of NFAs*

The programmable logic array (PLA) has been used as a systematic implementation of deterministic finite automata (see [11], e.g.). In these implementations the states are binary-coded, and the bits representing the new state are computed from the bits of the old state and the current inputs.

While we shall not attempt to describe the mechanics of PLAs in detail here, a rough idea of how they work can be obtained by looking ahead to Figure 7. There we see a typical PLA, which is a two-dimensional array of wires, divided vertically into an *and-plane* and an *or-plane*. Certain signals (labeled state $b, \ldots,$ state e in Figure 7) are fed back from the or-plane to the and-plane, with an implied delay of one time unit. New values of the feedback signals and output signals (L, C, and A in Figure 7) are computed in the following manner. Imagine signals with value 1 originating at the left end of each horizontal wire. In order for that signal to cross the and-plane, all the vertical wires that it intersects at a dot must have value 1. If the

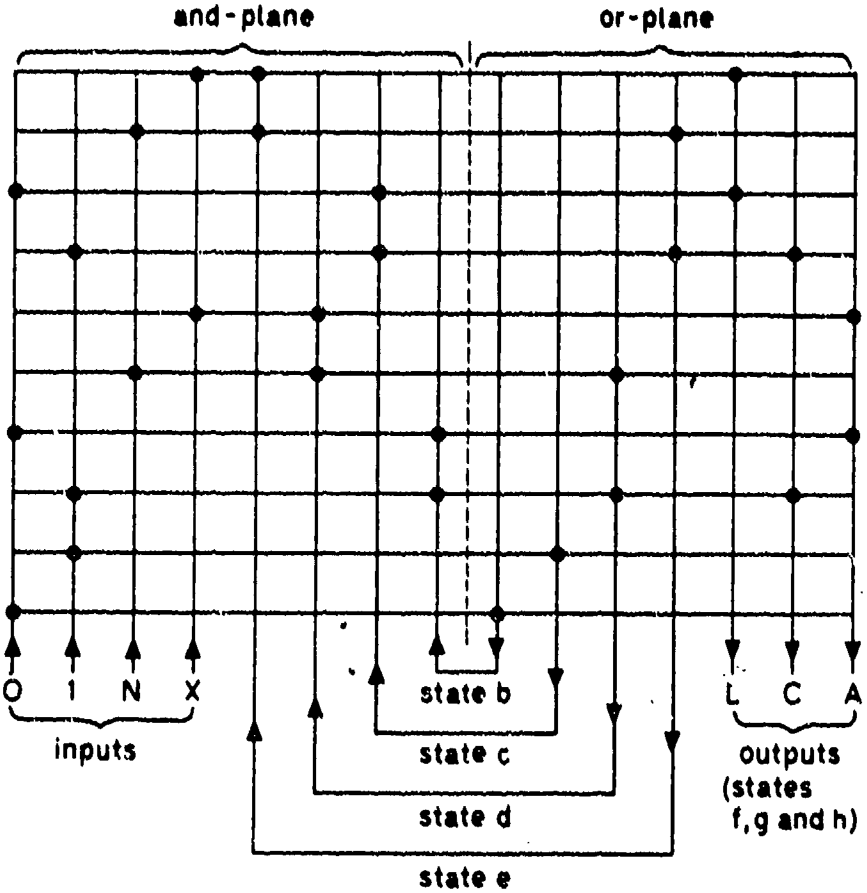


FIG. 7. PLA for machine of Figure 2.

signal reaches the right end of the and-plane, it enters the or-plane and makes 1 every vertical line it intersects at a dot. For example, in Figure 7, if the fourth and fifth vertical wires (X input and state e) are 1, the top wire will have its signal reach the or-plane and turn on the L output.

An alternative use of PLAs is to use one bit for each state of an NFA. We shall assume that the NFA has no ϵ-arcs, although that restriction is not essential if we allow states to be fed back from the or- to and-plane without any delay. For each arc of the NFA, labeled a and entering state q from state p, we create a term in the formula that tells whether q is one of the states in which the NFA is currently found.[2] This term is ap; that is, the term has the value true if and only if the input is a and state p was previously on. State q will be on at the next clock tick if and only if one of its terms has the value true; that is, there is some arc labeled a to q from a previously on state.

We may conclude from the above remarks that the number of rows of the PLA, each of which corresponds to a term in the formula for one or more states, is no greater than the number of arcs in the NFA.[3] The number of columns in the PLA is twice the number of states (for the next and previous versions of each state) plus the number of input bits and their complements, if needed.

Example 3. Let us implement the NFA of Figure 2 as a PLA. We begin by noticing that there are eight states, so in principle we need sixteen columns for the

[2] We shall say a state is *on* when the NFA is in that state.

[3] Recall that, technically, an arc with several labels is shorthand for a set of arcs, each with one label and the same source and destination.

	inputs				state code					
	0	1	X	N	A	C	L	S_1	S_2	S_3
a	b	c	a	a	0	0	0	1	0	0
b	d	e	a	a	0	0	0	0	0	0
c	d	e	a	a	0	0	0	0	0	1
d	d	e	a	a	1	0	0	0		
e	f	g	h	i	0	1	0	0		
f	d	e	a	a	0	0	1	0		
g	f	g	j	k	0	1	0	1		
h	b	c	a	a	1	0	0	1		
i	b	c	h	i	0	0	0	0	1	0
j	b	c	a	a	0	0	1	1		
k	b	c	j	k	0	0	0	0	1	1

FIG. 8. DFA from Figure 2 and one possible state encoding.

next and previous states. The inputs are coded by three bits, so we might assume we need six more columns. However, let us assume that the inputs are decoded into the four logical signals 0, 1, N, and X, by the table in Figure 1. We thus need a total of 20 columns. Furthermore, if we sum the numbers of labels on each of the arcs in Figure 2, we see that we apparently need 16 rows. However, we can do considerably better than this if we observe the following.

(1) States f, g, and h have no arcs out, and therefore their values need not be fed back, as those values are not used in the terms for any states. However, we must compute values for these states because they are final states. This arrangement saves three columns.
(2) State a is always on. Therefore, it need not be computed, and terms involving state a can use "true" in its place. This saves four rows and two columns.
(3) The transitions from b to d and g on input 1 require only one row, since the conditions are the same. Similarly, the two transitions from c on input 1 require only one row. Thus two additional rows can be saved.

The resulting PLA has 15 rows and 10 columns. It is shown in Figure 7, where circles represent connections. □

It is interesting to compare Figure 7 with the conventional PLA implementation of machines. If we convert the NFA of Figure 2 to a minimum-state DFA, we find the latter has 11 states. By way of comparison, we chose a particular encoding for states of this DFA. The encoding included the A, C, and L output bits and three other bits (the minimum necessary, since five of the states have $A = C = L = 0$). The state transition table and the encoding are shown in Figure 8.[4] Blanks in the state code entries indicate that either 0 or 1 may be used, that is, states with blank S_2 and S_3 entries have four alternative encodings, and we can use the most convenient one when one of these states is the next state.

Obviously, we could use only four bits to encode states, but then we would have to compute the output bits anyway, giving back some of the columns we saved by using shorter codes for the states, and also requiring additional terms to be computed, possibly increasing the number of rows required.

While we cannot be sure we have a minimum-row PLA, even after restricting ourselves to the state encoding of Figure 8, a careful selection of terms sufficient to compute the six state bits resulted in a 22 × 26 PLA. That is, there were 26 terms

[4] Note that the states in Figure 8 represent subsets of the states in Figure 2, and there is no necessary relationship between states of the same name in the two figures.

required, and the 22 columns consist of 6 for the next state, 12 for the previous state bits and their complements,[5] and 4 for the input bits.

The product of the dimensions for the conventional PLA implementation is about four times what it is for the NFA-based implementation. The inclusion of space around the peripheries of the two PLAs for drivers and clocking gates will reduce the 4 : 1 ratio somewhat, but there is still a clear advantage for the NFA approach to this design problem.[6]

We do not wish to generalize the results of one example to all sequential machine designs. Our method will be advantageous only when the problem at hand lends itself to a succinct description by regular expressions. For example, our methods do not work well on the traffic light example in [11], because that controller embodies a modulo four counter, and regular expressions are not convenient for expressing counts.

Let us summarize this section by formalizing the relationship between the size of regular expressions and the size of PLAs needed to implement them.

THEOREM 1. *For every collection of regular expressions consisting of a total of n operands, and with ϵ not in the language of any expression,*[7] *over an alphabet of at most 2^{i_0} symbols (i.e., i_0 bits are used to encode inputs), there is a PLA signaling the recognition of each of these expressions; this PLA has at most $2n$ rows and $2(n + i_0 + 1)$ columns.*

PROOF. First, let us assume there is but a single regular expression; the construction for multiple expressions will be clear from our discussion of single expressions. Begin by converting the expression to a nondeterministic finite automaton with ϵ transitions, using the McNaughton–Yamada construction of Figure 4. Observe from that construction that there will be exactly n states with transitions out that are labeled other than ϵ. These are exactly the states on the left of Figure 4c that were introduced by the n uses of that part of the construction. Call these n states, along with the initial and final states, *distinguished.*

Eliminate ϵ-arcs by computing for each distinguished state the set of distinguished states that it can reach along paths with exactly one non-ϵ-arc and any number of ϵ-arcs. Replace the existing arcs by an arc labeled a from distinguished state p to distinguished state q whenever there is such a path labeled a from p to q. By the definition of distinguished state and the fact that ϵ is not in the language of the expression (and therefore there is no ϵ-labeled path from initial to final state), the new NFA is equivalent to the old, and surely the new has no ϵ-arcs.

Our PLA will have $2i_0$ columns for the input bits and their complements. It has a column in the and-plane for the initial state, a column in the or-plane for the final state, and columns in both planes for the other distinguished states. Since there are n other distinguished states, $2(n + 1)$ columns are needed for states, and there are a total of $2(n + i_0 + 1)$ columns.

[5] Note that the PLA implementation of nondeterministic finite automata never requires the complements of state bits.

[6] One of the referees, and also M. Foster and H.-T. Kung, pointed out that in principle, since the PLA represents a deterministic finite automaton equivalent to the original nondeterministic finite automaton, it is never possible for the NFA approach to be superior to the conventional method. We believe, however, that the issue is not only the size of the theoretically optimal PLA, but the ease of finding such a PLA. It is doubtful that going from a DFA to a state encoding that reflects the underlying NFA is an easy problem.

[7] Strictly speaking, no PLA can recognize ϵ, since there is a clock tick that must elapse between an input and an output. However, a simple modification would allow direct (unclocked) connections from input and output, either through the PLA or around it.

The rows correspond to transitions in the new NFA. For each distinguished state p other than the initial and final states, we need one row that is turned on whenever p is on (i.e., the NFA is in state p, among others), and the input symbol is that for which p has transitions out. In the or-plane, this row turns on columns for the states reachable from p by the arcs labeled with that symbol. In this manner we create n rows.

We also need rows for each symbol labeling one or more transitions out of the initial state. These rows are turned on when the initial state wire is on and the input wires representing that symbol are on. It is easy to see that the number of such symbols is no greater than n, since each transition out of the initial state must correspond to a non-ϵ transition of the original NFA. All transitions are either from the initial state or from a distinguished state that is neither initial nor final. Hence, all transitions are implemented by the above rows, and thus a total of at most $2n$ rows are needed in our PLA. □

As a consequence of Theorem 1, for fixed input alphabet Σ, we can implement regular expressions of length n in $O(n^2)$ area. However, in practical examples it is common for an n-state NFA to be converted to a DFA with roughly n states. For example, the 8-state NFA of Example 1 becomes an 11-state DFA. If that is the case, then an n-state NFA might be implemented by an $O(\log n) \times O(n)$ PLA. The DFA-based implementation of machines would be superior, but, as mentioned, it is not necessarily easy to find the best PLA for a DFA. Surprisingly, we shall see in the next section that there is a totally different approach to the implementation of regular expressions that yields a circuit of dimensions $O(\sqrt{n}) \times O(\sqrt{n})$.

6. *A Hierarchical Implementation of Regular Expressions*

An inspection of Figure 4, which shows how to construct an NFA from a regular expression, suggests that we could lay out a circuit directly on a chip, if we represent states of the NFA by appropriate logical elements, represent ϵ-arcs by wires, and represents arcs labeled by input symbols by wires with gates checking for that symbol. This approach was suggested in [12], for example. The states used in Figure 4 can be divided into two classes:

(1) those that have ϵ-arcs out, and
(2) those that do not, that is, they are final states or have arcs leaving that are labeled by an input symbol.

States in the second group are implemented by *latches*, that is, pairs of inverters connected in a loop, with one clock phase to control the output of each. Those in the first group are really nothing more than junction points in the circuit, allowing two signals to merge (through an or-gate) or one signal to fan out into two identical signals (no logic at all is needed here).

When building large circuits from smaller ones, it helps if we view each circuit as a rectangle, as suggested in Figure 9. Power is supplied at the upper left and passed to the right if needed by a circuit to the right. Ground drains at the lower right, and ground from circuits to the left is passed in at the left if necessary.[8] Two phases of a clock and the bits needed to represent an input symbol are passed in at the left and will pass out at the right if needed to supply a circuit to the right.

There is a signal called *state-in* that, if it is 1, turns the start state of the circuit on at phase one of the clock. An output signal, called *state-out*, is turned on at clock

[8] Viewing power and ground this way enables us to avoid crossing them, which is generally not feasible in integrated circuits.

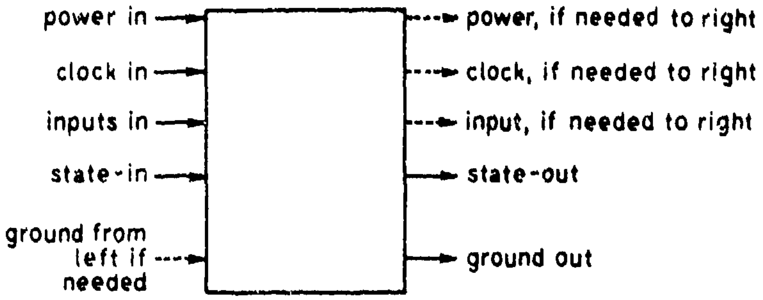

power in
clock in
inputs in
state-in
ground from left if needed
power, if needed to right
clock, if needed to right
input, if needed to right
state-out
ground out

FIG. 9. Format of a circuit implementing a regular expression.

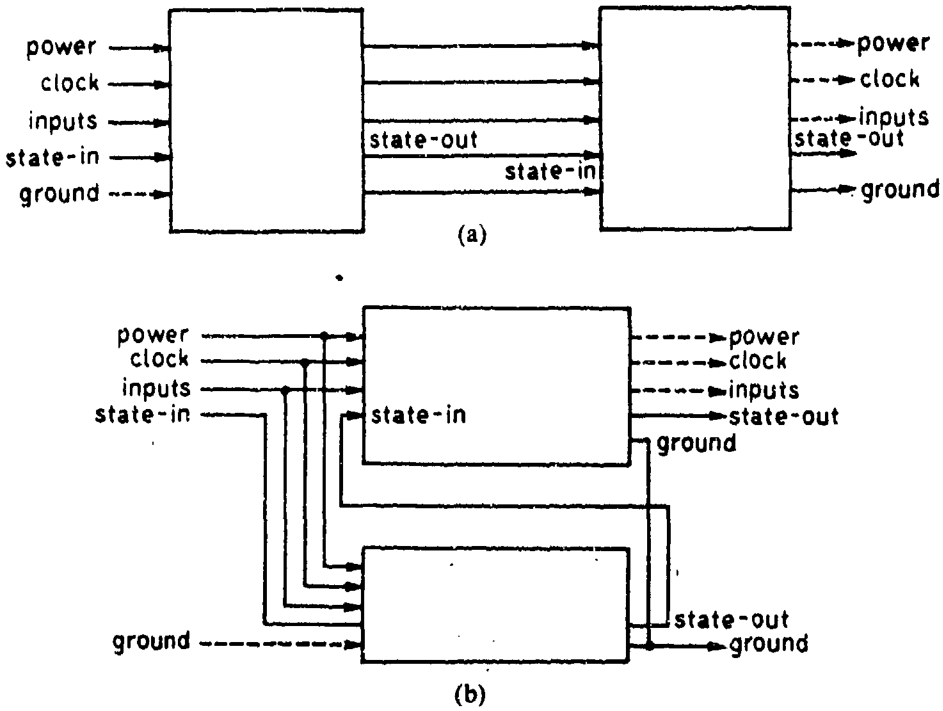


FIG. 10. Circuit connections. (a) Horizontal connections. (b) Vertical connections.

phase two if the circuit enters its accepting state. In general, phase one of the clock is used to decide which states will be on after processing the current input symbol and to propagate this information through states with ϵ-arcs leaving. Phase two is used to transfer the decisions made at phase one to the output of the latches that do not have ϵ-arcs out.

Let us suppose we have circuits for regular expressions R_1 and R_2, and we wish to construct a circuit for $(R_1)(R_2)$. We can connect the circuits in cascade as suggested by Figure 4e; this connection is shown in Figure 10a. Note that the final state of the first machine is given an ϵ-arc out. Thus the latch representing it is no longer needed or appropriate. We must replace it by a junction point or, if there are several input arcs for that state, by an or-gate. As latches can be expected to require more area than a single gate or junction point, we can make this replacement without worrying about the geometry of the circuit, and we shall henceforth assume such changes are made when necessary, not only in the concatenation construction, but in the union and closure constructions as well.

Figure 10b shows an alternative organization for the circuit, in which the first machine is placed above the second. Similarly, when we implement the union construction of Figure 4d, we can choose to place either constituent circuit above the other or place either to the left of the other. The closure construction, since it does

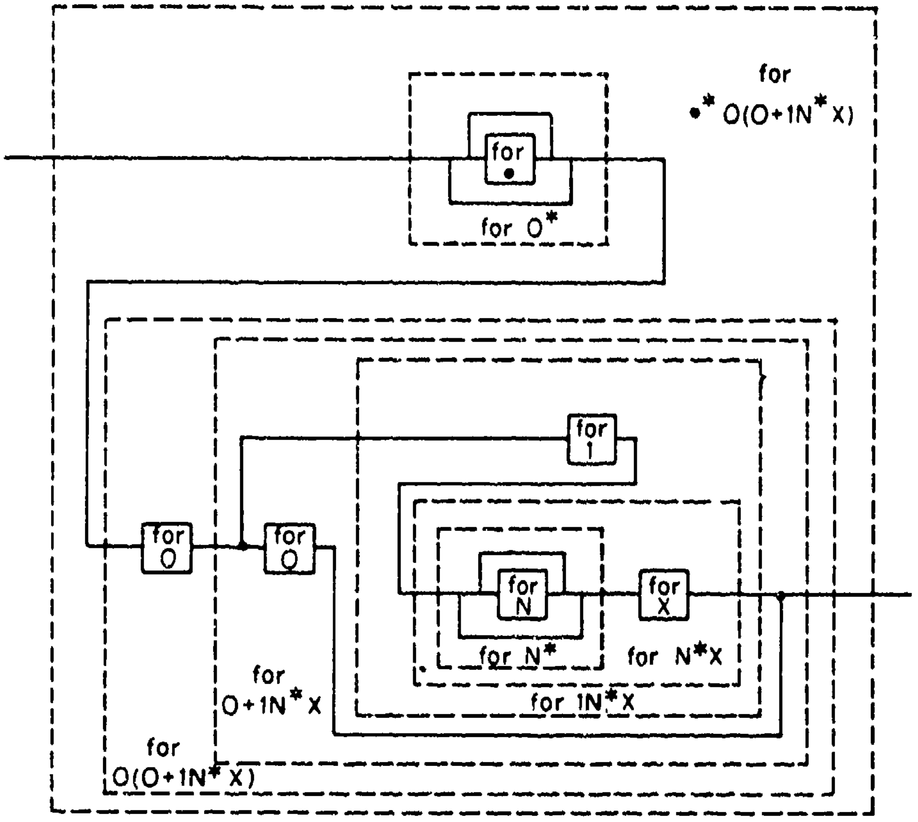


FIG. 11. Straightforward implementation of the McNaughton-Yamada algorithm.

not combine two circuits, gives us little choice; we must simply augment the circuit with surrounding feedback and feed-forward wires as suggested by Figure 4f.

The reason we care about the relative positioning of circuits is that we desire each circuit to have an *aspect ratio* (ratio of height and width) near one. For example, if we must combine two circuits that are longer than they are high, we would prefer the vertical connection of Figure 10b to the horizontal connection of Figure 10a, since the former has a squarer shape than the latter. The reason, in turn, for desiring an aspect ratio near one is that, on the average, we can couple squarish circuits with less waste space than we can couple elongated circuits. For example, neither Figure 10a nor b is very good if one of the constituent circuits is very tall and thin while the other is short and wide. Another motivation for keeping aspect ratios low is that the basic circuits, such as latches, cannot be designed in a fixed area with a fixed aspect ratio if the area allotted is small and the aspect ratio is high. Thus the rectangles representing the basis constructions of Figure 4a–c must be allocated space of limited aspect ratio.

Unfortunately, just keeping the aspect ratio within bounds is not sufficient to guarantee efficient use of space, for one of two constituent circuits could be significantly larger than the other. For example, an expression like

$$(\cdots((a_1 + a_2)a_3 + a_4)a_5 + \cdots)a_n$$

forces us to create either a long, thin circuit with many long wires or an L-shaped circuit, if we restrict ourselves to the constructions of Figure 10. As another example,

$$(((a_1^* a_2)^* a_3)^* \cdots a_n)^*$$

requires n nested feedback loops, so it appears to require $O(n^2)$ space no matter what we do.[9] As we shall see, all these problems can be solved, and circuits for these expressions, taking area that is proportional to the length of the expressions, can be generated automatically. Before proceeding to the techniques involved, let us illustrate the basic McNaughton–Yamada construction and also show how the combination of unequally sized circuits tends to waste space.

Example 4. Let us build a circuit for the regular expression $.^*0(0 + 1N^*X)$, whose parse tree was given in Figure 3. Using judicious choices between horizontal and vertical connections when union and concatenation constructions are used, we might obtain the layout[10] suggested by Figure 11. There, only state-in and state-out wires are shown; input, power, ground, and clock wires are omitted. □

7. *A Compact Hierarchical Implementation of Regular Expressions*

There are three insights necessary to our implementation of regular expressions. First, we must observe that given any regular expression whose parse tree has $n \geq 2$ leaves, we can find a subtree that has more than $n/3$ but no more than $2n/3$ leaves. For example, the tree of Figure 3 has six leaves, and its subtree for expression $1N^*X$ has three leaves, which is greater than two and no greater than four. The subtree for $0 + 1N^*X$ would also qualify. This application of "divide and conquer" to binary trees was first used in [9].

Once we have found a subtree of about half the leaves, we can build a circuit C_1 for it, and we can build a circuit C_2 for the remaining tree, with a dummy leaf in place of the deleted subtree. This leaf is an imaginary input symbol, and when applying the McNaughton–Yamada algorithm to it, we generate a start state s and a final state f, using the construction of Figure 4c, but without the arc. A wire connects state s of C_2 to the start state of circuit C_1, and another wire runs from the final state of C_1 to f. In effect, we have simply removed C_1 from its rightful place between s and f. Note that both states s and f are unnecessary and can always be replaced by junction points, even if latches are created for them initially. The arrangement is sketched in Figure 12.

Notice how, if C_1 and C_2 are about the same size and shape, they are likely to fit together, either side by side, as shown, or one above the other. In comparison, if we had to distort C_2 by "squeezing" C_1 between s and f, we might or might not achieve a compact layout.

As our circuit design rules introduced in Section 2 do not permit us to cross more than three wires at a point, simply laying down the wires shown in Figure 12 could lead to an illegal circuit. We must therefore "pull apart" C_1 and C_2 at four *channels*, in which the wires can run. The idea is shown in Figure 13. To create a channel, we select a line across the circuit. Circuit elements and wires running parallel to the line are held at one side of the line, while wires perpendicular to the line are stretched. After stretching some constant amount, there will be room to fit another wire parallel to the line. Circuit elements to which the wire must be connected are, we presume, crossed by the line and can be moved into the channel to connect with the wire.

The above method for creating channels will be successful if the original circuit

(1) has all wires running horizontally or vertically;

[9] Note, however, that there is an equivalent regular expression with an $O(n)$ area circuit.

[10] We shall use the term "layout" in what follows to refer to the relative positioning of various subcircuits. The term does not have its more usual connotation of a much more detailed design. However, the positionings we use are intended to be such that a layout, in the usual sense, could be done without repositioning.

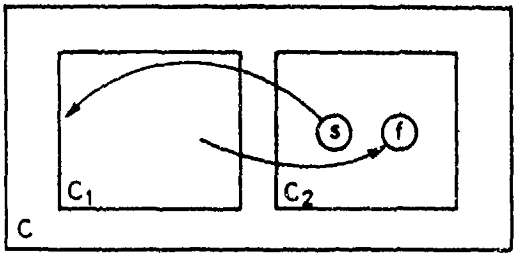


FIG. 12. Divide-and-conquer implementation of regular expressions.

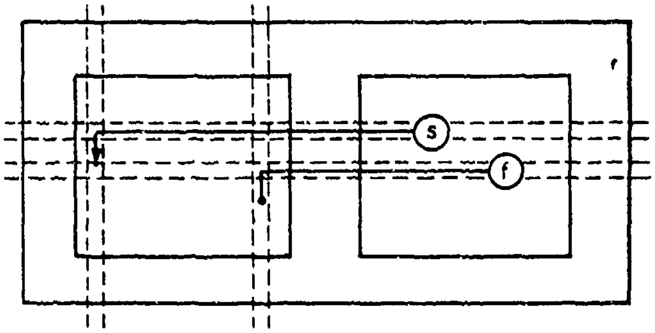
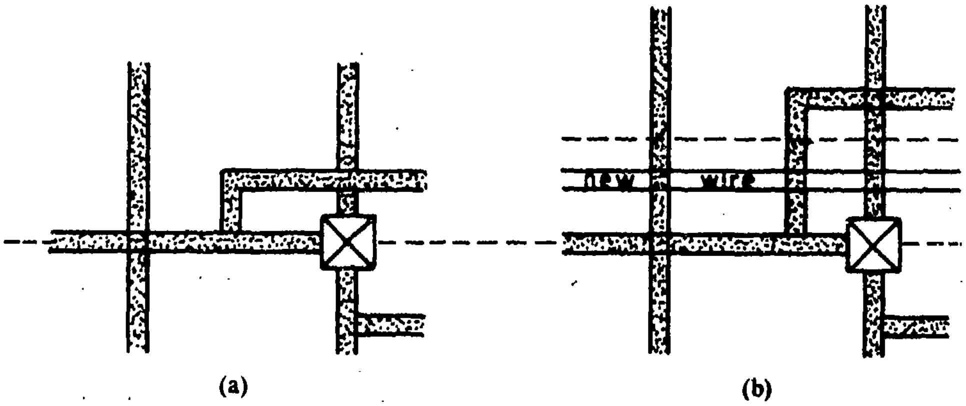


FIG. 13. Channels to carry wires of Figure 12.



(a) (b)

FIG. 14. The channel creation process (a) before creating channel and (b) after creating channel.

(2) never has more than two wires crossing at a point; and
(3) uses "circuit elements" from some fixed set, so there is an a priori bound on the size of a circuit element.

Condition (3) guarantees that a channel of some fixed width will be sufficient to run a new wire without crossing any circuit elements, and (1) and (2) assure that the new wire will only cross one other wire at a time. Figure 14 gives an example of the channel creation process.

The second insight needed is that even if C_1 and C_2 are about the same size, their aspect ratios and relative sizes might be such that they do not have a common dimension, either the same width (for a vertical arrangement as in Figure 12) or the same height (for a horizontal arrangement). Unless our recursive circuit layout algorithm works in such a way that when applied to C_1 and C_2 we can expect a dimension in common, we may be forced to connect C_1 and C_2 in a manner that wastes about a quarter of the space. Since the waste can go on at every level of the recursion, we shall have an algorithm that uses area $n^{\log_2(8/3)} \approx n^{1.41}$ to implement a

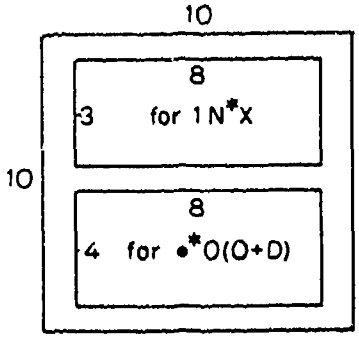


FIG. 15. Initial layout.

regular expression of size n. This result is superior to obvious methods, but not as good as we can do.

The solution to the above problem is to design our recursive layout algorithm to take as parameters

(1) the parse tree of the expression for which we want to design a circuit,
(2) the number of nodes of that tree; and
(3) the desired aspect ratio, a real number in the range $\frac{1}{4}$ to 4.

We assert that there is a constant d such that for each parse tree of $n \geq 2$ leaves, there is a circuit of aspect ratio r and area dn, for any r in the range $\frac{1}{4} \leq r \leq 4$. It will be shown that for $n \geq 3$ and aspect ratio r between $\frac{1}{4}$ and 4, we can always arrange C_1 and C_2, either horizontally or vertically, with a border and channels adequate for connections between C_1 and C_2 and to the "outside world," and with this arrangement, recursive calls to design C_1 and C_2 can be given appropriate aspect ratios between $\frac{1}{4}$ and 4, so that C_1 and C_2 will have a side in common.

Example 5. Let us consider how the parse tree of Figure 3 would be processed recursively by the circuit layout algorithm. First, we must find a node from which between $\frac{1}{4}$ and $\frac{3}{4}$ of the leaves descend. The preferred candidate is the root of the subtree for $1N^*X$, which divides the leaves into two equal parts. As the initial call to the circuit routine would normally ask for a square circuit (aspect ratio 1:1), we may position the subcircuits for $.^*0(0 + D)$ (footnote 11), and $1N^*X$ either horizontally or vertically; let us choose the latter. As the first expression has four leaves and the second has three,[12] the heights of the two subcircuits should be in the ratio 4 to 3. They are given the same width. A sample arrangement, in which the entire circuit is allocated a 10 × 10 area (in some units), and borders are one unit wide, is shown in Figure 15.

We now lay out the circuits for $1N^*X$ and $.^*0(0 + D)$ in the rectangles of aspect ratios 3:8 and 1:2, respectively. We should, in principle, divide each of these expressions into two parts and recursively synthesize their circuits from circuits for the parts. However, we omit the details of those recursive calls. One circuit that could result is shown in Figure 16. □

The third necessary insight is that two or more consecutive applications of the closure operator are equivalent to one. That is, for any regular expression R we have $(R)^* = ((R)^*)^*$. As a consequence, we may eliminate superfluous *'s and view regular expressions as if all the operators were binary operators chosen from the list: union, concatenation, union-then-closure, and concatenation-then-closure. We use the con-

[11] D stands for the particular dummy symbol used as a placeholder for the expression $1N^*X$.

[12] However, if we are careful, we can avoid allocating circuit area for the dummy symbol, which we know will be represented in the circuit by junction points only.

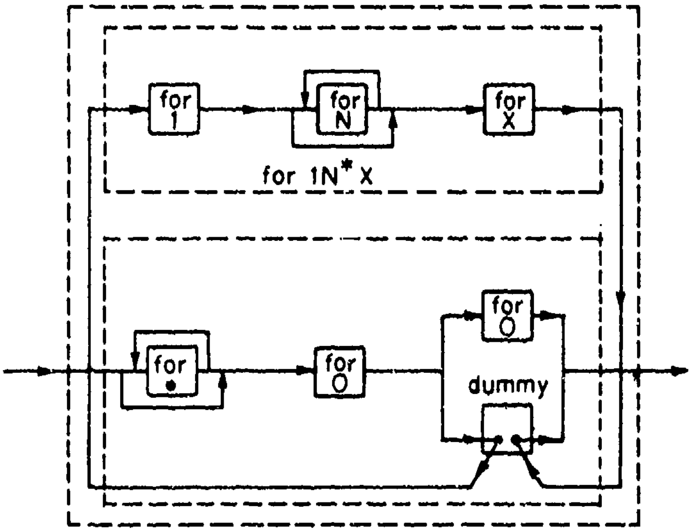


FIG. 16. Complete layout for the expression of Figure 3.

structions of Figure 4d and e, followed by f, when closure is desired, to build circuits, just as in the McNaughton–Yamada algorithm. Operands are either single symbols or symbols to which closure is applied, and circuits for operands can be constructed by Figure 4a–c optionally followed by f. Note that this algebraic simplification is necessary to avoid awkward situations like the expression $a^{**\cdots*}$ which, if the McNaughton–Yamada algorithm were applied blindly, would result in a circuit of area $O(n^2)$.

The heart of the circuit layout algorithm is the recursive procedure LAYOUT sketched in Figure 17. The algorithm itself is a call to LAYOUT(T, n, 1), where T is the parse tree for a regular expression of length n, that is, T is assumed to have n leaves. In LAYOUT, Σ is assumed to be a fixed input alphabet defined globally, so its size may be regarded as constant. Also, b is a constant chosen large enough that the total width of the channels and border area, either in the horizontal or vertical direction, is bounded above by b. Note that channels need to carry one wire each, while border areas may need to carry $4 + \log_2 \|\Sigma\|$ wires, one for each of the input bits, and one each for power, ground, and the two clock phases. Finally, $A(n)$, the area allotted to a circuit for a regular expression of length n, is a function of the form $dn - e\sqrt{n} - f$, whose adequacy we shall show in the next section.

8. Analysis of the Algorithm

We now show that LAYOUT can be made to use $O(n)$ area, by showing that a linear function $A(n)$ can be chosen. We must pick $A(n)$ to satisfy the following constraints.

(1) The area $A(n)$ available for C in Figure 14 must not exceed area $A(n_1) + A(n_2)$ used for C_1 and C_2 plus the area needed for borders and channels.

(2) The aspect ratios of C_1 and C_2 must be in the range $\frac{1}{4}$ to 4 if that of C is.

(3) $A(2)$ must be large enough that we can build a circuit for any regular expression of length 2 in that area, with any aspect ratio up to 4.

LEMMA 1. *If $A(n) \geq 25b^2$ and C of Figure* 14 *has aspect ratio* 4 *or less, then C_1 and C_2 have aspect ratios in the range $\frac{1}{4}$ to* 4.

PROOF. The extreme cases we must consider are when $n_1 = 2n_2$ (footnote 13) and either

[13] Since $2n/3 \geq n_1 > n/3$ and $n_1 + n_2 = n + 1$, it is easy to show that n_1/n_2 must be in the range $\frac{1}{2}$ to 2.

```
function LAYOUT($T$, $n$, $r$);  {$T$ is a parse tree with $n \geq 2$ leaves. LAYOUT returns
  a circuit of area $A(n)$ and aspect ratio $r$; we assume without loss of generality that
  $r \geq 1$; otherwise rotate the layout 90°. The circuit returned has only horizontal and
  vertical wires, and at no point do more than two wires overlap.}
begin
    if $n = 2$ then
        use McNaughton-Yamada algorithm to produce a circuit $C$
    else  {$n \geq 3$}
    begin
        select a node $N$ of $T$ such that $N$ is the root of a subtree with $n_1$ nodes,
          where $n/3 < n_1 \leq 2n/3$;
        let $T_1$ be the tree with root $N$;
        let $T_2$ be $T$ with the subtree rooted at $N$ replaced by a dummy leaf;
        $n_2 := n - n_1 + 1$;  {$T_2$ has $n_2$ leaves}
        {now we perform horizontal decomposition, as in Figure 12}
        $h := \sqrt{A(n)/r}$;  {$h$ is the height of circuit $C$ of Figure 12}
        $h_1 := h - b$;  {height of $C_1$ and $C_2$ in Figure 12}
        $w_1 := (h * r - b) * n_1/(n_1 + n_2)$;  {width of $C_1$}
        $w_2 := h * r - b - w_1$;  {width of $C_2$}
        $r_1 := w_1/h_1$; $r_2 := w_2/h_2$;  {aspect ratios for $C_1$ and $C_2$}
        $C_1$ := LAYOUT($T_1$, $n_1$, $r_1$);
        $C_2$ := LAYOUT($T_2$, $n_2$, $r_2$);
        Separate circuits $C_1$ and $C_2$ to make two horizontal and four vertical
          channels for their interconnections, as shown in Figure 13. Figure 14
          showed how this operation could be done in such a way that wires could
          be laid along the channels without violating the circuit design rules we
          have assumed;
        Add border around $C_1$ and $C_2$, and run wires for inputs, etc., to feed both
          circuits and to produce wires out of the bottom and right edge, as
          indicated in Figure 10;
        Call the resulting circuit $C$;
    end;
    return $C$;
end
```

FIG. 17. The recursive procedure LAYOUT.

(a) C has aspect ratio 1, in which case C_2 could be too tall and narrow, or
(b) C has aspect ratio 4, in which case C_1 could be too short and wide.

Let the height of C be h. Then in case (a) the height of C_2 is $h - b$ and its width is $(h - b)/3$, so its aspect ratio is 3, satisfying the lemma. In case (b) the height of C_1 is again $h - b$ and its width is $\frac{2}{3}(4h - b)$. Thus its aspect ratio will be 4 or less provided

$$h - b \geq (\tfrac{1}{4})(\tfrac{2}{3})(4h - b),$$

that is, $h \geq \frac{5}{2}b$. Since the area of C in this case is $4h^2$, the lemma follows. □

THEOREM 2. *There exist positive constants d, e, and f such that for all $n \geq 2$, the function LAYOUT will succeed in producing a circuit if the allotted area $A(n)$ is $dn - e\sqrt{n} - f$.*

PROOF. Let us, for the moment, assume that d, e, and f satisfy the lemma for $n = 2$. Notice that when we divide a tree T of $n \geq 3$ leaves into T_1 and T_2 in function LAYOUT, neither n_1 nor n_2 can be 1. Thus we can attempt to prove by induction, with a basis of $n = 2$, that area $A(n) = dn - e\sqrt{n} - f$ suffices for LAYOUT to produce a circuit. To develop the induction, let $n \geq 3$ and $n_1 = an$ for some constant a, $\frac{1}{3} < a < \frac{2}{3}$. Then the areas of C_1 and C_2 are $A(an)$ and $A((1 - a)n + 1)$, respectively, since $n_1 + n_2 = n + 1$. Observe that C_1 and C_2 are, by Lemma 1, of limited aspect ratio, and their areas are also chosen by LAYOUT to be of limtied aspect ratio. Hence the borders and channels in Figure 14 have area that is proportional to any side of C_1 or C_2, the constant of proportionality naturally depending on which side is chosen. Specifically, there is some constant c such that the extra area of circuit C, beyond that of C_1 and C_2, is at most $c\sqrt{A(an)}$. Thus

$$A(n) = \max_{1/3<a\leq 2/3} \left[A(an) + A((1-a)n + 1) + c\sqrt{A(an)}\right]. \tag{1}$$

We assume $A(m) \leq dm - e\sqrt{m} - f$ for $2 \leq m < n$, and show that the same holds when $m = n$. By (1) it suffices to show that

$$dn - e\sqrt{n} - f$$
$$\leq \max_{1/3<a\leq 2/3} \left[adn - e\sqrt{an} + (1-a)dn + d - e\sqrt{(1-a)n + 1} - 2f + c\sqrt{adn}\right]. \tag{2}$$

In the last term of (2), $\sqrt{A(an)}$ has been conservatively replaced by $\sqrt{adn}$. Simplifying (2), we obtain

$$0 \geq \max_{1/3<a\leq 2/3} \left[e\sqrt{n} - e\sqrt{an} - e\sqrt{(1-a)n + 1} + d - f + c\sqrt{adn}\right]. \tag{3}$$

Dividing (3) by $-e\sqrt{n}$ yields

$$0 \leq \max_{1/3<a\leq 2/3} \left[\sqrt{a} + \sqrt{1 - a + \frac{1}{n}} - 1 + \frac{f-d}{e\sqrt{n}} - \frac{c}{e}\sqrt{ad}\right]. \tag{4}$$

The first three terms on the right of (4) sum to at least 0.39. The next term can be made 0 if we pick $f = d$. The last term is no more than 0.28 if we choose $e = 3c\sqrt{d}$. Thus, for these choices of e and f in terms of d, (4) is satisfied; hence so is (2).

Now we must satisfy the condition that $A(2)$ is adequate to hold all circuits for regular expressions of length 2. We simply observe that we can pick d so that $A(2) = d - 3c\sqrt{d}$ exceeds any quantity we choose, so an adequate value of d can be found. □

One may wonder if the linear bound on area for a general regular expression is the best that could be achieved. We believe it is, because of another assumption that is generally made ([2], e.g.) about integrated circuits, that there is a finite (as opposed to infinitesimal) amount of area needed to store one bit of information. If that is the case, then we cannot improve on the linear growth rate in Theorem 2, because there are regular expressions of length proportional to n that require n bits of information to be remembered if we are to recognize them. A simple example is the family

$$(0 + 1)^*1(0 + 1)(0 + 1)\cdots(0 + 1),$$

where n terms $(0 + 1)$ follow the 1. For each n, this regular expression denotes the set of strings of 0's and 1's that have a 1 n positions from the end. Clearly, we must remember the last n inputs if we are to recognize all strings in the language.

9. *Implementation Considerations*

A compiler for single regular expressions has been implemented by J. Ullman; it follows many of the ideas outlined here. The major point of departure is that no slicing of circuits is attempted. Rather, for each dummy symbol it creates a new input wire. The circuit to "recognize" the dummy symbol simply switches the dummy input wire and the state wire, with no clocking gate.

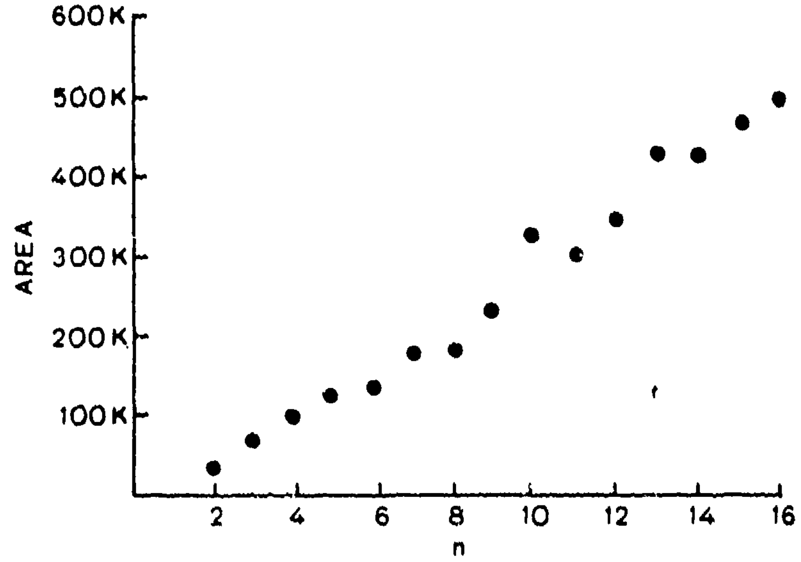


FIG. 18. Growth of circuit size with expression size.

As different parts of the circuit have different numbers of input wires, and the number of input wires, including dummies, can get large, we do not know for certain that this approach yields linear-sized circuits in the worst case. However, empirically, the growth rate is fairly linear. Figure 18 shows the area of circuits generated by the compiler for the regular expressions S_n that are intended to express the condition that n input wires are fired in sequence.

By way of explanation, the input to the compiler distinguishes between "wires" and "symbols," the former being physical wires and the latter abstract symbols mentioned by the regular expression. Thus we may define symbols a_i and b_i for $1 \leq i \leq n$, to represent the fact that wire i is on or off, respectively. Then S_n can be written

$$b_1 + .^*(a_1b_2 + a_2b_3 + \cdots + a_{n-1}b_n).$$

S_n then recognizes those strings that violate the property that all n wires are on in turn. The number of operands in S_n is actually $2n$, rather than n, but the difference should be of no concern.

The compiler in question also implements the PLA strategy on small subexpressions. It was found that the smallest ratio of area to number of operands for our PLA implementation occurs when the number of operands is about 8, and approximates $6000\lambda^2$ per operand.[14] However, since there is overhead in wiring rectangles together, our best results occur when PLAs are built for expressions in the 10–15 operand range, sometimes more. The circuits whose sizes are represented in Figure 18 were created before the PLA feature was implemented. That feature, used with various size ranges for expressions that are to be implemented as PLAs, results in a 10–40 percent reduction in circuit area, typically.

Straightforward circuits to perform the same function as the expressions S_n are quite a bit smaller than the circuits produced by the compiler. However, it is unlikely that the compiler can compete with the best ad-hoc circuits in any case. A more promising example concerns regular expressions we shall call P_n that recognize whether the first n synbols in a string of 0's, 1's, and "don't cares" match the last n, with "don't care" matching anything. P_n has $O(n \log n)$ symbols, $n(6 + \log_2 n)$ to be

[14] λ is the basic unit of measurement in integrated circuit design. In 1980, circuits could be fabricated with λ in the range 2-3 microns, but as technology improved, we expected λ to shrink and hence the same circuits to take substantially less area.

exact. While ad-hoc circuits based on shift registers exist and are probably smaller than what the compiler can produce for all values of n, a fairer comparison would be with a PLA mechanically generated from a state diagram, say using the Xerox modules described in [11]. We have produced for P_{16} a circuit of about two million λ^2, which is close to the area of the straightforward PLA. Since the area of the PLA will grow as n^2, naturally we may expect the compiler to look progressively better for values of n above 16.

10. *Related Work*

The ideas of divide-and-conquer layout and of channel creation were also used independently by C. E. Leiserson [7] and by L. Valiant [15]. We could have used the results of [7] to show Theorem 2 by proving a "2-separator theorem" for the graphs of nondeterministic finite automata that we obtain by the McNaughton–Yamada construction. Strictly speaking, the connections needed for supplying, input, power, and so on, must be ignored in that theorem and handled outside the framework of [7].

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design*. Addison Wesley, Reading, Mass., 1977.
2. BRENT, R.P., AND KUNG, H.T. The chip complexity of binary arithmetic. Proc. 12th Ann. ACM Symp. on the Theory of Computing, Los Angeles, Calif., May 1980, pp. 190–200.
3. EHRENFEUCHT, A., AND ZEIGER, P. Complexity measures for regular expressions. *J. Comput. Syst. Sci. 12*, 2 (Apr. 1976), 134–146.
4. GRAY, J.P. Introduction to silicon compilation. 16th Design Automation Conf. Proc., June 1979, pp. 305–306.
5. HOPCROFT, J.E., AND ULLMAN, J.D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, Mass., 1979.
6. JOHANNSEN, D. Bristle blocks, a silicon compiler. 16th Design Automation Conf. Proc., June 1979, pp. 310–313.
7. LEISERSON, C.E. Area-efficient graph layouts (for VLSI). Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science, Oct. 1980, Syracuse, N.Y., pp. 270–281.
8. LESK, M.E. LEX—A lexical analyzer generator. Tech. Rep. CSTR-39, Bell Laboratories, Murray Hill, N.J., 1976.
9. LEWIS, P.M. II, STEARNS, R.E., AND HARTMANIS, J. Memory bounds for the recognition of context-free and context-sensitive languages. Proc. IEEE 6th Ann. Symp. on Switching Circuit Theory and Logical Design, Oct. 1965, pp. 191–202.
10. MCNAUGHTON, R., AND YAMADA, H. Regular expressions and state graphs for automata. *IEEE Trans. Comput. C9*, 1 (Mar. 1960), 39–47.
11. MEAD, C., AND CONWAY, L. *Introduction to VLSI Systems*. Addison Wesley, Reading, Mass., 1980.
12. MUKHOPADHYAY, A. Hardware algorithms for nonnumeric computation. *IEEE Trans. Comput. C28*, 6 (June 1979), 384–394.
13. SIEWIOREK, D.P. A survey of research on synthesis, evaluation, and automation of digital systems at CMU. Dep. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., 1979.
14. THOMPSON, C.D. Area-time complexity for VLSI. Proc. 11th Ann. ACM Symp. on the Theory of Computing, Atlanta, Ga., May 1979, pp. 81–88.
15. VALIANT, L. Universality considerations in VLSI circuits. *IEEE Trans. Comput. 30*, 2 (Feb. 1981), 135–140.

RECEIVED APRIL 1980; REVISED APRIL 1981; ACCEPTED OCTOBER 1981

AREA AND DELAY PENALTIES FOR RESTRUCTURABLE VLSI ARRAYS

(DRAFT--not for wide distribution)

Jonathan W. Greene *   *Abbas El Gamal* **

Abstract: The asymptotic penalties of restructuring homogeneous VLSI arrays for yield enhancement are investigated. Each element of the fabricated array is assumed to be defective with independent probability p. A fixed fraction R of the elements are to be connected into a prespecified regular pattern with no defects. The probability of successfully connecting the pattern must be bounded away from zero as its size increases. Let d be the length of the longest connection and t be the number of wiring tracks needed to accomplish the interconnection. It is shown that: (1) Connecting a chain of K elements from a linear array of N elements requires $d=\Omega(\log N)$ and $t=1$ track running parallel to the array. (2) Connecting a linear array of K fixed I/O ports to distinct non-defective elements from a parallel array requires $d=\Omega(\log N)$ and $t=\Omega(\log N)$. (3) Connecting K elements from an N-element linear array to K from a parallel N-element array, in pairs, requires only constant d and t. (4) Connecting a chain of K elements from an $N\times N$ array requires constant d and one track between elements; this problem is closely related to the percolation problem of statistical physics. In all the above cases, algorithms achieving the given bounds on d and t are presented which connect the array with probability approaching one. The algorithms run in $O(K)$ time. (5) Connecting a $K\times K$ square lattice from an $N\times N$ array is shown to require $d>\Omega(\sqrt{\log N})$. It is conjectured that only a constant number of tracks are required between elements.

* Stanford University. Work partially supported by an NSF Graduate Fellowship and under Air Force contract F 49620-79-C-0058.

** Stanford University. Work partially supported under DARPA contract MDA 903-79-C-0680 and Air Force contract F 49620-79-C-0058.

1. Introduction

The use of redundancy to maintain the manufacturing yield of VLSI circuits at economic levels is becoming a widespread practice.. For example, many producers of 64K RAMs use spare rows and columns. This has been reported to improve the yield by a factor of five to eight [1]. Lincoln Labs [2] and McDonnell Douglas [3] are currently experimenting with restructurable whole-wafer processor arrays. After testing the array elements, programmable links are either opened [1][4], closed [5], or switched [3] so as to connect the nondefective elements into the desired configuration.

The resulting improvement in yield is achieved at the expense of an increase in the number of elements on the chip, the addition of links and extra interconnections, an increase in signal delay and the effort of restructuring. In this paper we investigate these penalties for regular VLSI arrays. Our approach is best illustrated through the following simple example.

A linear array of K identical processors, connected in a chain, is to be implemented on a single integrated circuit, or chip. Assume that each processor has an independent probability p of being *defective* and $1-p$ of being *active*. Then the probability that a chip is functional is $(1-p)^K$ and the expected fraction of functional chips, or yield, approaches zero exponentially as K increases.

In order to prevent the yield from approaching zero, the number of processors on the chip is increased to $N=K/R$ for some $R<1-p$ and switches are provided to insert the processors in the chain. After manufacture, the processors are tested. If the number of nondefective, or active, processors is less than K the chip is discarded. Otherwise, K processors may be connected as shown in Figure 1.1. Since $R<1-p$, the probability that the chip will have sufficient active processors approaches one exponentially as K increases.

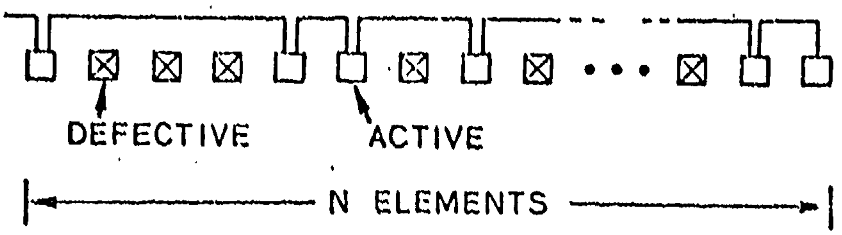


Fig: 1.1: Connection of a chain from an N-element linear array.

Unfortunately, signals from one processor to another may now encounter additional propagation delay since the connections between processors are longer than before. Suppose that the maximum tolerable connection distance is fixed at d, where each processor has unit width. Let X be the number of ways that a chain of K active processors can be connected under this constraint. There are fewer than N places at which a chain can start, and probability p^d that there will be too many defects before the next active processor. The probability that the chain can be connected is less than the expected value $EX < N(1-p^d)^K < N\exp(-Kp^d)$ which approaches zero unless the size of the chip, N, grows exponentially with K--a very unsatisfactory situation.

We will assume here and throughout that the fraction of elements connected, R, must be held constant as N grows. If we fix $R<1-p$, it is easily shown that $d=O(\log N)$[†] will suffice to connect a chain with probability approaching one exponentially in N.

In the following sections, we consider one and two dimensional arrays and several connection patterns. Except for the case of connection into a lattice (Sec. 5), specific linear time algorithms are given for restructuring the chip. These algorithms will, with probability approaching one, connect any fraction $R<1-p$ of the total number of elements. Our results are:

[†] We use $O(\cdot)$ to denote an upper bound, $\Theta(\cdot)$ to denote an exact bound, and $\Omega(\cdot)$ to denote a lower bound, all to within a constant factor.

Sec. 2: The connection of a linear array of $K=RN$ fixed input/output ports to distinct, active elements from a parallel N-element array is to be accomplished by means of a channel containing t wiring tracks between the ports and the array, with a switch at each crosspoint. A connection may not run along the channel for a distance greater than d. We term this arrangement, shown in Figure 1.2, a *selector*. Theorem 1 states that unless $t=\Theta(\log N)$ and $d=\Theta(\log N)$, the yield approaches zero. This is caused by the high probability that a run of $\Theta(\log N)$ consecutive defects exists within the array. The proof of Theorem 2 describes a scheme which achieves these bounds with yield approaching one polynomially in N.

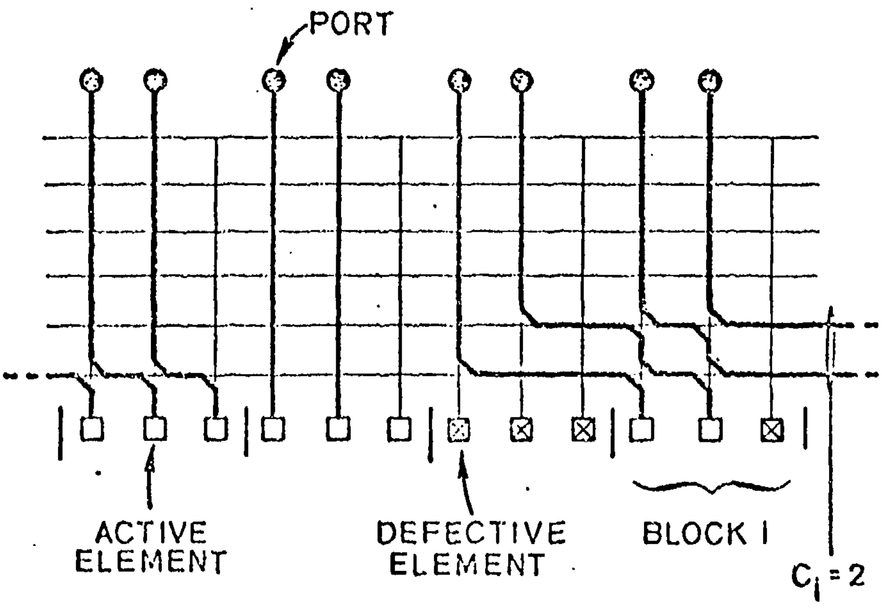


Fig. 1.2: A section of a selector. $K=(2/3)N$ and six wiring tracks are provided. A typical block and cut C_i are shown; these will be defined in the proof of Theorem 2.

Sec. 3: The connection of K pairs of active elements from two parallel N-element arrays, shown in Figure 1.3, is surprisingly easier than the task of the selector. A run of defects does not necessarily cause a problem here because there are no fixed ports--alternate pairs of elements may be connected from other parts of the array. The proof of Theorem 3 describes a scheme with $d=t=O(1)$ which achieves yield approaching one exponentially in N.

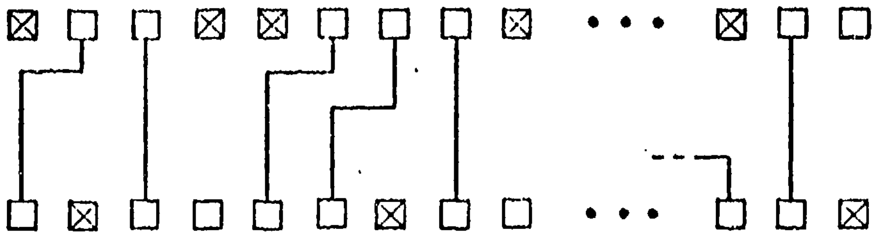


Fig. 1.3: Pairwise connection of two parallel N-element linear arrays.

Sec. 4: The connection of a chain from a two-dimensional array, as shown in Figure 1.4, can be achieved with $t=1$ wiring track between elements, $d=O(1)$ and yield approaching one polynomially in N. This is demonstrated by Theorem 4, which is based on certain results concerning the percolation problem of statistical physics. (A survey of percolation theory will be found in [6]). A slightly different scheme achieving similar results has been found independently by Leighton and Leiserson [7].

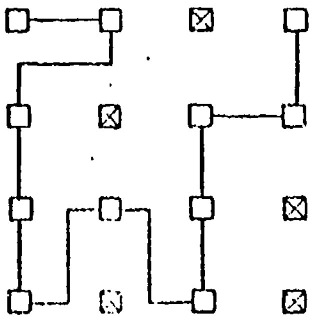


Fig. 1.4: Connection of a chain of K=11 elements from a 4×4 array.

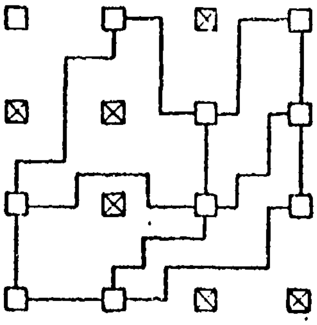


Fig. 1.5: Connection of a 3×3 square lattice from a 4×4 array.

Sec. 5: The connection of a $K \times K$ square lattice from an $N \times N$ array, as shown in Figure 1.5., requires a maximum wire length $d=\Omega(\sqrt{\log N})$ to prevent yield from approaching zero. This is demonstrated in Theorem 5 using a lemma concerning the nonseparability of the square lattice. We conjecture that only a constant number of wiring tracks between elements is necessary, and show that if this is true $d=O(\sqrt{\log N})$ in fact suffices to connect the lattice with probability approaching one polynomially in N for any $R<\bar{p}$. We note that Leighton and Leiserson [7] have proposed a connection scheme with $d=O(\sqrt{\log N}\log\log N)$ and $t=O(\log\log N)$.

The problems of connecting a chain and a lattice from a two-dimensional array of faulty elements were first studied by Manning [8] and Aubusson and Catt [9]. They proposed algorithms for connecting chains, trees and lattices, but were unable to provide any theoretical analysis of their work.

2. Selectors

We begin by proving a lower bound on the maximum connection distance and number of tracks required in a selector.

Theorem 1: For any $0<\delta<1$, the probability that $K=RN$ ports, aligned parallel to a linear array of N elements, can be connected to distinct active elements tends to zero as $O(p^{N^{1-\delta}/\delta\log N})$ unless the number of tracks in the channel t and the maximum connection distance d both satisfy

$$d,t > \left\lfloor \frac{\delta R \log N}{-2\log p} - \frac{1}{2} \right\rfloor.$$

Proof: For any $0<\delta<1$, let

$$m = \left\lfloor \frac{\delta R \log N}{-\log p} \right\rfloor.$$

Divide the array and the ports into $\lfloor NR/m \rfloor$ blocks each containing m ports by cuts perpendicular to the array. Any extra piece of the selector is ignored. Let n_i be the number of elements in block i. Since the array has N elements,

$$\sum_{i=1}^{\lfloor NR/m \rfloor} n_i \le N. \tag{2.1}$$

Suppose there is a block with all its elements defective. If $2d+1\le m$ or $2t+1\le m$ the middle port of the block cannot be connected to an active element. Under these constraints on t and d, the probability that a connection exists is bounded by

$$\begin{aligned}
P_{con} &\le P\{\text{no block has all elements defective}\} \\
&= \prod_{i=1}^{\lfloor NR/m \rfloor} (1-p^{n_i}) \\
&\le \left(1-p^{m/R}\right)^{\lfloor NR/m \rfloor} \qquad \text{under constraint (2.1)} \\
&< \exp(-p^{m/R}\lfloor NR/m \rfloor) \\
&< \exp(-p^{\delta\log N/\log p}(N(-\log p)/\delta\log N-1))
\end{aligned}$$

$$\approx O(p^{N^{1-\delta}/\delta \log N}).$$

Substituting the definition of m into the assumptions on d and t yields the result.▪

(We note that Theorem 1 may be extended to the case where the selector is laid out within a convex region with all the ports on the boundary).

It is easily shown that $d,t = O(\log N)$ will suffice to connect the selector with probability approaching one. Simply divide the selector into $N/c\log N$ blocks of $c\log N$ elements and $Rc\log N$ ports for some properly chosen constant c. The Chernoff bound may be applied to show that the probability that a given block has fewer than $Rc\log N$ active elements approaches zero exponentially in $\log N$. Even when multiplied by the number of blocks, this value still approaches zero. Thus with probability approaching one, *all* blocks have at least enough active elements to connect their ports. Each block may then be connected separately using $t = Rc\log N$ tracks and maximum connection distance $d = c\log N$.

Not surprisingly, the constant c is fairly large for this simple scheme. In the proof of the next theorem we propose a better scheme which is nearly as easy to implement, though more difficult to analyze. For values of R near $\bar{p}$, the constant is one to two orders of magnitude smaller. For example, if $\bar{p}=0.9$ and $R=7/8=0.875$, the simple scheme requires $c \approx 308$ while the scheme presented below requires only $c \approx 2.4$. Simulation results for these values of $\bar{p}$ and R and three fixed values of N will be presented below.

Theorem 2: For any rational number $r < 1-p$, let $w > 0$ be any constant such that

$$p\exp(wr) + (1-p)\exp(-w(1-r)) < 1. \qquad (2.2)$$

Then for any $0 < \delta < 1$ it is possible to connect $K = rN - O(\log N)$ ports to distinct active elements of a linear array with probability $1 - O(N^{1-1/\delta})$, using a maximum connection distance

$$d = \left\lceil (rw\delta)^{-1}\log N \right\rceil$$

and number of tracks

$$t = \lceil rd \rceil .$$

The fraction of elements connected, $R = r - O(N^{-1}\log N)$, approaches r.

Proof: We first verify that w exists for $r<1-p$. Let $f(w) = p\exp(wr)+(1-p)\exp(-w(1-r))$, the expression in (2.2). Note that $f(0)=1$. Evaluating the derivative of $f(w)$ at $w=0$ we find

$$\begin{aligned}\frac{d}{dw}f(0) &= pr-(1-p)(1-r)\\ &< 0\end{aligned}$$

since $r<1-p$. Thus there must be some $w>0$ such that $f(w)<1$.

Now we describe the selector. For the given r choose the smallest integer b such that rb is an integer. We assume that $N>>b$ and divide the array into $\lfloor N/b \rfloor$ blocks of b elements. There are fewer than b elements left over; these are ignored. Locate ports above the first rb elements of each block, as is shown in Figure 1.2 for the case $r=2/3$, $b=3$. The other elements of each block will serve as "spares." Let d and t be defined as in the statement of the theorem. A port may be connected to the element below it or one of the next d elements by means of the wiring tracks and crosspoints as shown. This arrangement requires at most t wiring tracks and a maximum connection distance d.

The connection procedure is as follows: starting from the left, connect each port to the leftmost, previously unused, active element among those $d+1$ to which the port can be connected.

Let G_i denote the number of ports in blocks 1 through i which must be connected to subsequent blocks (see Figure 1.2). The procedure can fail only in one of the following circumstances:

(A) All $d+1$ elements to which some port can be connected are either defective or have already been used. Because the procedure gives priority to the leftmost unconnected port, the next $\lfloor rd \rfloor$ ports are also, as yet, unconnected. Assume the last element within distance d of the port is in block i. Then, allowing for active spares in block i, $C_i \geq 1+\lfloor rd \rfloor-(1-r)b > rd-(1-r)b$.

(B) End effect: the last $\lceil rd \rceil$ ports may not be able to connect to their full set of d elements since the array ends before their distance constraint runs out. By not employing these ports, which reduces K by only $O(\log N)$, we can ignore this problem.

Thus if there is no $C_i \geq rd-(1-r)b$, $1 \leq i \leq \lfloor N/b \rfloor$, the procedure will successfully connect the ports.

Let $X_i = rb -$(number of active elements in block i). We have the following recursion on C_i:

$$C_0 = 0$$
$$C_i = \max \{ C_{i-1}+X_i, 0 \}.$$

Let $S_m = \sum_{i=1}^{m} X_i$. Since the X_i are independent and identically distributed one can show [10] that C_i is distributed the same as

$$\max \{ 0, S_1, S_2, \ldots, S_k \}. \qquad (2.3)$$

We apply the Chernoff bound to $P(S_i \geq rd-(1-r)b)$. For any $X \in \{X_i\}$ let

$$\begin{aligned}\Phi(w) &= E\exp(wX) \\ &= \exp(wrb) \left[p+(1-p)\exp(-w)\right]^b \\ &= \left[p\exp(wr)+(1-p)\exp(-w(1-r))\right]^b .\end{aligned}$$

Note that $\Phi(w)<1$ by (2.2).

The probability that the procedure fails,

$$\begin{aligned}P_{fail} &\leq P\left[C_i \geq rd-(1-r)b \text{ for some } i\right] \\ &\leq \sum_{i=1}^{\lfloor N/b \rfloor} P\left[C_i \geq rd-(1-r)b\right]\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{\lfloor N/b \rfloor} P\left[\max_{0 \le k \le i} S_k \ge rd-(1-r)b\right] && \text{by (2.3)} \\
&\le \sum_{i=1}^{\lfloor N/b \rfloor} \sum_{k=1}^{i} P\left[S_k \ge rd-(1-r)b\right] \\
&\le \sum_{i=1}^{\lfloor N/b \rfloor} \sum_{k=1}^{i} \exp(-w(rd-(1-r)b))\Phi^k(w) && \text{(Chernoff bound)} \\
&= \exp(-w(rd-(1-r)b)) \sum_{i=1}^{\lfloor N/b \rfloor} \Phi(w)\,\frac{1-\Phi^i(w)}{1-\Phi(w)} \\
&\le \exp(-w(rd-(1-r)b)) \left[\frac{N}{b}\,\frac{\Phi(w)}{1-\Phi(w)} - \left(1-\Phi^{\lfloor N/b \rfloor}(w)\right)\left(\frac{\Phi(w)}{1-\Phi(w)}\right)^2\right] \\
&= O(N^{1-1/\delta})
\end{aligned}
$$

for the given value of d. This completes the proof.▪

The connection procedure employed in the proof of Theorem 2 is clearly suboptimal since a port cannot be connected to elements on its left. An improved scheme which connects each port in turn to the leftmost unused, active element within the distance and track constraints is harder to analyze asymptotically, but can be used for simulations. The results of some simulations using the bidirectional scheme are shown in Figure 2.1. We note that both the suboptimal and improved procedure require $O(N)$ steps.

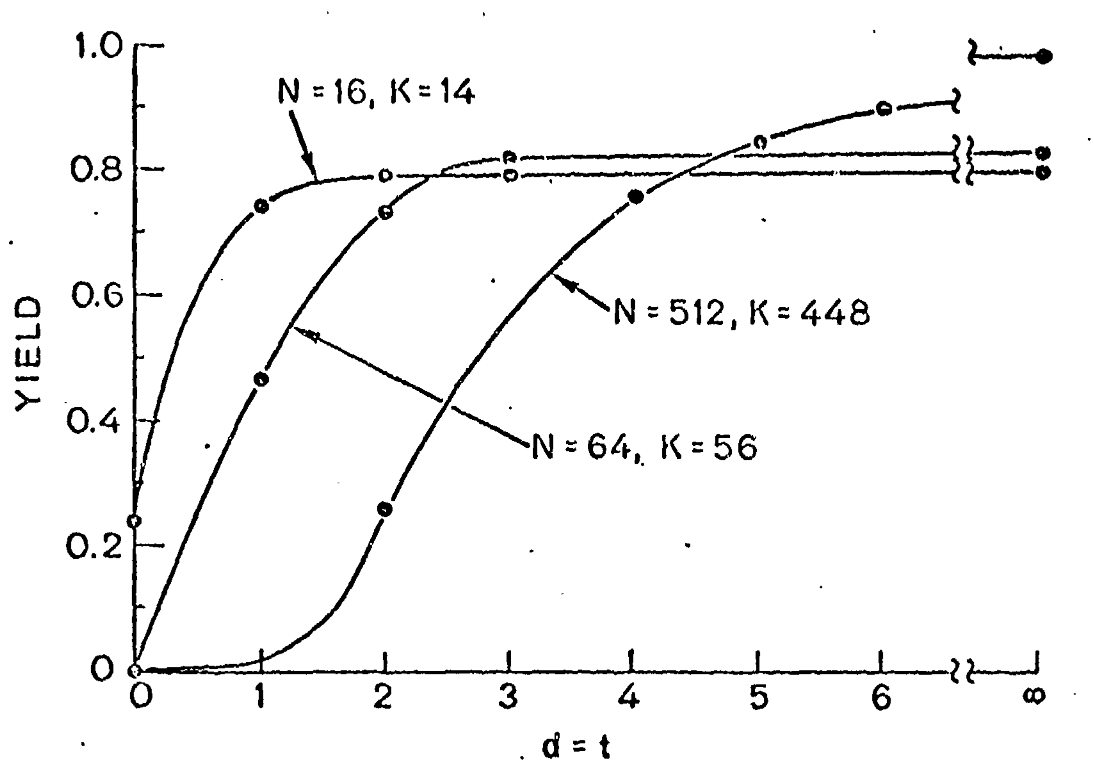


Fig. 2.1: Yield over 1000 trials for an N element selector with p =0.1, R=7/8 and d=t as indicated on the horizontal axis.

3. Pairing of Two Parallel Arrays

We now demonstrate that parallel connection of two linear arrays requires only constant d and t.

Theorem 3: For any $r<\bar{p}=1-p$ and $R<r$ it is possible to connect $K=RN$ pairs of active elements from two N-element arrays with probability approaching one exponentially in N using a constant number of tracks and constant maximum distance

$$d,t = \lceil p\bar{p}\,(\bar{p}-r)^{-2}/2\rceil.$$

Proof: We propose a scheme under which the expected fraction of elements that may be connected is r.

Let

$$b = \lceil p\bar{p}\,(\bar{p}-r)^{-2}/2\rceil$$

Divide two linear arrays into blocks b elements wide and provide $t=b$ tracks between them. Choose any block of one array and let X be the number of active elements in the block and let Y be the number of active elements in the facing block of the other array. No matter where defects occur in the block, we can connect $\min\{X,Y\}$ pairs of active elements with the tracks provided and maximum distance $d=b$.

Since X and Y are independent and both binomially distributed with parameters $[b,\bar{p}]$,

$$\begin{aligned} E^2(|X-Y|) &\le E(X-Y)^2 \\ &= E(X-EX)^2 - 2E(X-EX)E(Y-EY) + E(Y-EY)^2 \\ &= 2\ Var(X) = 2bp\bar{p}. \end{aligned}$$

Then the expected number of pairs which can be connected in each block is

$$\begin{aligned} E(\min\{X,Y\}) &= EX-E(\max\{0,X-Y\}) \\ &= EX-(1/2)E(|X-Y|) \\ &\ge b\bar{p} - \sqrt{bp\bar{p}/2} \\ &\ge br. \end{aligned}$$

The expected fraction of elements which can be connected is thus at least r. Since the defects in each block are independent and $R<r$ the Chernoff bound may be applied to show that, with probability $1-\exp(-cN)$ for some constant $c>0$, a fraction R of the elements can be connected.■

4. Chains Connected From a Two-dimensional Array

The problem of connecting a chain of active elements from a two-dimensional array is closely related to percolation theory. Percolation processes have been studied extensively since they were first defined by Broadbent and Hammersley [11].

The site percolation problem concerns an infinite lattice of sites which are empty with some independent probability q and occupied with probability $\bar{q}=q-1$. A site is said to *percolate* if it is a member of or adjacent to an infinite cluster of occupied sites. Broadbent and Hammersley demonstrated that the probability of a site percolating is the same for any site, and so may be expressed as a percolation probability function $R(\bar{q})$, which is monotonic increasing and attains the value 1 at $\bar{q}=1$. They also showed that $R(\bar{q})=0$ for $\bar{q}$ less than some critical value characteristic of the lattice.

Little else of an analytical nature is known about $R(\bar{q})$, although Monte Carlo estimates have established empirical curves for various lattices [12]. The curve for a square lattice is reproduced in Figure 4.1.

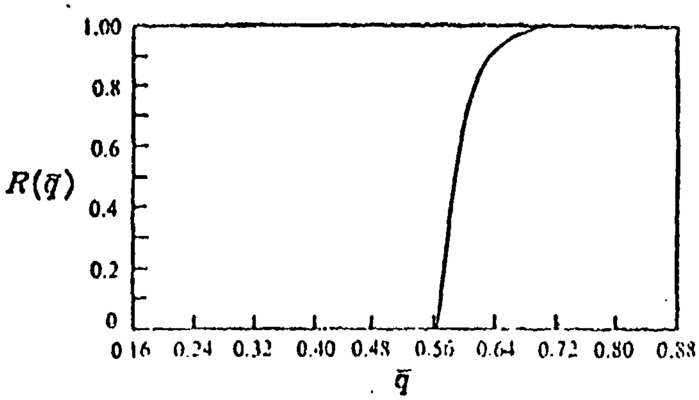


Fig. 4.1: The percolation function $R(\bar{q})$ for the square lattice as determined by Monte Carlo estimates. (From Fig. 6 of Frisch, Hammersley and Welsh [12]).

Our scheme for connecting a chain from an array may be analyzed using some results in percolation theory, contained in the following four lemmas. We restrict consideration to a square lattice, though the results readily generalize to others.

A site percolates unless it is enclosed by empty sites. An *enclosing walk* is defined to be a closed self-avoiding walk on empty sites; diagonal as well as horizontal and vertical steps are permitted since such a walk is capable of enclosing a site. Lemma 1 bounds the probability that a site will be enclosed, and thus the percolation probability.

Lemma 1: For a square lattice, if $q<(1/7)\exp(-2/3)\approx 0.0733$, then $R(\bar{q})\geq 1-(945/4)q^3$.

Proof: An enclosing walk starting from a given site has 8 choices for the second site and no more than 7 choices for each subsequent site. Thus there are fewer than $(8/49)7^L$ distinctly shaped enclosing walks of length L. Furthermore, there are no more than $(L^2/8-L/2+1)$ translations of a walk of a certain shape which enclose a given site. Each of the L sites in the walk must be empty.

The probability that a given site is enclosed by an enclosing walk of length at least $L_0\geq 4$ is no more than the expected number of such walks, which by the above arguments is less than

$$\sum_{L=L_0}^{\infty}(8/49)7^L(L^2/8-L/2+1)q^L$$

$$\leq \sum_{L=L_0}^{\infty}(1/49)(7q)^L L^2$$

$$< (1/49)\int_{L_0-1}^{\infty}(7q)^x x^2 dx$$

$$= (1/49)(7q)^{L_0-1}[-9/\log(7q)+6/\log^2(7q)-2/\log^3(7q)]$$

$$<(135/28)(7q)^{L_0-1}.$$

The second inequality is valid because for $q<(1/7)\exp(-2/3)$ the terms of the series decrease monotonically for $L \geq 3$.

Since an enclosing walk must contain at least 4 sites, $1 - R(\tilde{q}) \leq (135/28)(7q)^3$, which yields the result.▪

We now prove a convergence result for the fraction of sites that percolate.

Lemma 2: Let X be the number of sites in an $N \times N$ section of the infinite square lattice that percolate. If $q < (1/7)\exp(-2/3)$, then for any $r < R(\tilde{q})$, $P(X \leq rN^2) \leq O(N^{-2})$.

Proof: Let A and B denote the event that site a and site b percolate, respectively. Let $d(a,b)$ be the Manhattan (rectilinear) distance between a and b. Let W denote the event that there is an enclosing walk surrounding a or b of length at least $d(a,b)/2$, and W^c its complement. Note that there can be no overlap between a walk of length less than $d(a,b)/2$ enclosing a and a walk of length less than $d(a,b)/2$ enclosing b. Thus $P(A,B) = P(A,B,W^c) = P(A,B \mid W^c)P(W^c)$ $= P(A \mid W^c)P(B \mid W^c)P(W^c) \leq P(A)P(B)/P(W^c) = P(A)P(B)\left[1 + P(W)/P(W^c)\right]$.

In the proof of Lemma 1 it was demonstrated that the probability that a given site is enclosed by a walk of length at least L_0 is less than $c_1(7q)^{L_0}$ for some constant c_1. Thus $P(W) < 2c_1(\sqrt{7q})^{d(a,b)}$. We can therefore choose constants d_0 and c_2 such that if $d(a,b) \geq d_0$ then $P(W^c) \geq c_2$.

Finally, note that the number of sites at distance $d > 0$ from a given site on a square lattice is $4d$. We can now upper bound the variance of X as follows.

$$\begin{aligned}
Var(X) &= \sum_a \sum_b P(A,B) - \left[\sum_a P(A)\right]^2 \\
&\leq \sum_{\substack{a \\ d(a,b)<d_0}} \sum_b P(A,B) + \sum_{\substack{a \\ d(a,b)\geq d_0}} \sum_b P(A)P(B)\left[1 + P(W)/P(W^c)\right] - \left[\sum_a P(A)\right]^2 \\
&\leq \sum_a \sum_{d=0}^{d_0-1} (4d) + \sum_a \sum_{d=d_0}^{N} (4d)\, 2c_1(\sqrt{7q})^d / c_2 \\
&= O(N^2)O(1) + O(N^2)O(1) = O(N^2).
\end{aligned}$$

The expectation of X is $EX=N^2R(q)$. By Chebyshev's inequality,

$$\begin{aligned} P(X\le rN^2) &\le P(|X-EX|\ge EX-rN^2) \\ &\le \frac{Var(X)}{(EX-rN^2)^2} \\ &= O(N^{-2}).\blacksquare \end{aligned}$$

Lemma 3: Suppose $q<1/7$. Consider those sites within an $N\times N$ section of the infinite lattice which are members of infinite clusters of occupied sites. Except for a fraction $O(N^{-1}\log N)$, these sites form a single cluster within the $N\times N$ section, with probability $1-O(N^{-2})$.

Proof: It is known that on an infinite lattice, the set of occupied sites contains only one infinite cluster, with probability 1. (Proved in [13] Sec. 9 for bond percolation, extended to site percolation in [14]). However, when the $N\times N$ section is removed from the infinite lattice, the part of the infinite cluster lying within the section may be disconnected into several components, separated by self-avoiding walks on empty sites (not necessarily closed). By arguments similar to those in the proof of Lemma 1, it is easily shown that the expected number of self-avoiding walks on empty sites starting at a site within the $N\times N$ section and having length at least $L_{\max}=-4\log(N)/\log(7q)$ is $O(N^{-2})$. Thus with high probability, only sites within $L_{\max}$ of the boundary of the section can be cut off from the rest of the cluster. These sites account for a fraction $O(N^{-1}\log N)$ of the N^2 sites.$\blacksquare$

Lemma 4: For each $i=1,\ldots,n$, let A_i be the event that every site in some finite non-empty set is occupied. For each $j=1,\ldots,m$ let B_j be the event that every site in some finite non-empty set is occupied. No assumption about the exclusivity of the sets is made. Let $A=\cup A_i$ and $B=\cup B_j$. Then $P(A|B)\ge P(A)$.

The proof of this intuitive lemma will be found in [13, Lemma 4.1].

We are now ready for the main result of this section.

Theorem 4: With probability $1-O(N^{-2})$, a chain of length $K=RN^2$ can be connected from an $N \times N$ array with maximum distance

$$d = \left\lceil \sqrt{3\log((\tilde{p}-R)/c)\ /\ \log(p)} \right\rceil,$$

for some constant $c>0$, and with one track in each channel.

Proof: The general idea is as follows. Group the elements into N^2/b square blocks of b elements each. Choose b so that each block has high probability of containing at least 4 active elements.

Each block may be considered as corresponding to a site on a square lattice, and if the block has at least 4 active elements, consider the site occupied. Using the previous lemmas, we show that nearly all sites are in or adjacent to a single large cluster of occupied sites.

A tree of maximum degree 4 can be constructed which spans the cluster of occupied sites and all sites adjacent to the cluster, with all non-leaf nodes situated on occupied sites. This can also be considered as a spanning tree on the blocks. Since all "non-leaf" blocks have at least 4 active elements, a chain of active elements may be formed by looping around the tree without ever having to connect two elements from non-adjacent blocks. (See Figure 4.2). The maximum Manhattan connection distance required is

$$d \leq 3\sqrt{b} - 2. \tag{4.1}$$

Only one track is needed between elements to accomplish this. The construction of the spanning tree requires $O(K)$ steps and the connection of subchains in the blocks also requires $O(K)$ steps since there are only a constant number of elements in each block.

For any choice of b, let q be the probability that a particular block contains fewer than 4 active elements. Then

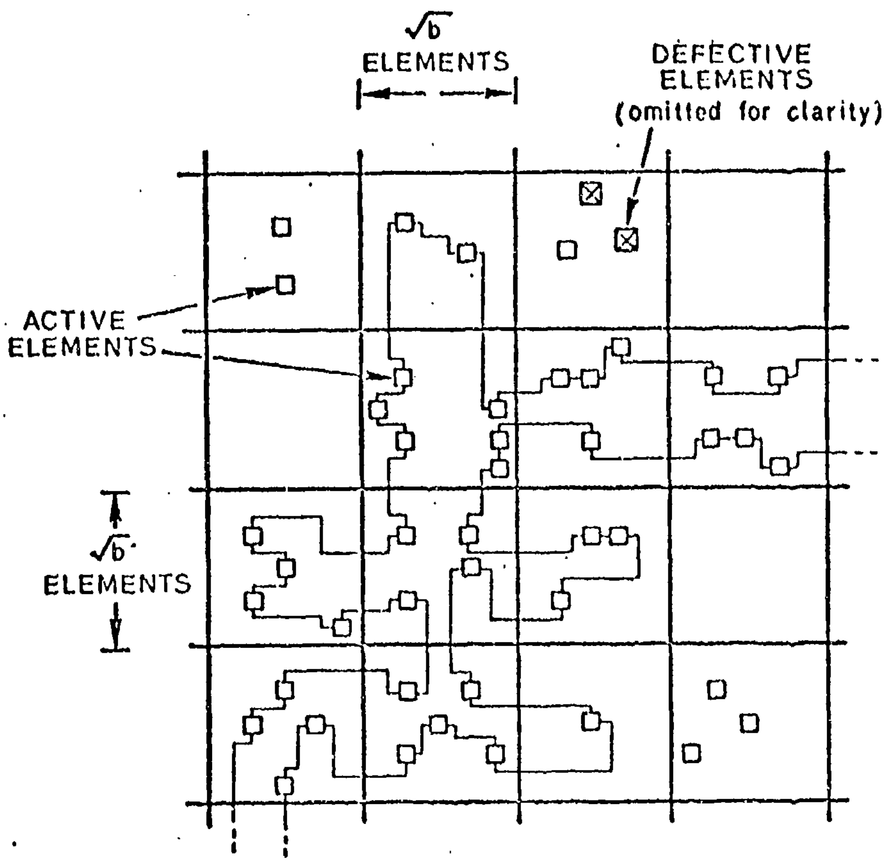


Fig. 4.2: A section of an array connected into a chain. Each block contains b elements, although only the active elements are shown.

$$q = \sum_{i=0}^{3} \begin{bmatrix} b \\ i \end{bmatrix} \bar{p}^i p^{b-i} \le 2b^3 p^{b-3}. \tag{4.2}$$

Choose b so that $q<(1/7)\exp(-2/3)$. By (4.2) this requirement is satisfied by $b=O(\log^{-1}p)$. For this value of q, we can apply Lemmas 1-3 to percolation on an $N/\sqrt{b} \times N/\sqrt{b}$ lattice of sites corresponding to blocks, as described above.

By Lemmas 2 and 3, for any

$$r < R(\bar{q}) \tag{4.3}$$

at least rN^2/b sites will be members of or adjacent to a single cluster, with probability $1-O(N^{-2})$. By the correspondence of sites to blocks, this means that at least rN^2/b blocks are members of or adjacent to a single cluster of blocks containing 4 or more active elements.

We proceed to bound the number of active elements in any such rN^2/b blocks. Choose any

$$R < r\bar{p}. \tag{4.4}$$

By a simple application of Lemma 4, the probability that the number of active elements in the blocks is at least RN^2 given that all the blocks are members of or adjacent to a cluster is at least as great as the unconditional probability, where the elements of the blocks are considered independent. Since $R<r\bar{p}$, the Chernoff bound proves that the unconditional probability is $1-O(N^{-2})$.

By Lemma 1,

$$\begin{aligned} R(\bar{q})\bar{p} &\geq [1-(945/4)q^3]\bar{p} \\ &\geq [1-(945/4)(2b^3p^{b-3})^3]\bar{p} && \text{by (4.2)} \\ &\geq \bar{p} - cp^{d^2/3} \end{aligned} \tag{4.5}$$

by (4.1) for some constant $c>0$.

Combining (4.3)-(4.5), we have shown that for any R such that $R<\bar{p}-cp^{d^2/3}$, or equivalently

$$d > \sqrt{3\log((\bar{p}-R)/c)\ /\ \log(p)},$$

a chain of RN^2 elements can be connected with probability $1-O(N^{-2})$.▪

The following example illustrates the practicality of this scheme. Suppose each element is defective with probability $p=0.5$. Choose $b=9$, and hence $d=7$. The probability that a block of 9 elements has at least 4 active elements is $\bar{q}\approx.7461$. From Figure 4.1, we see that for an infinite square lattice and this value of q, any block is practically certain to be a member of or adjacent to a cluster. Even for a finite 30×30 element array, and accordingly a 10×10 lattice of blocks, an average of 94.9 blocks were members of or adjacent to the largest cluster of blocks with 4 or more active elements in 500 random trials. (Std dev.≈8.5).

We conclude this section by mentioning the problem of external connections to the chain, or actually the loop, of elements formed by the scheme. If two input/output pins are placed in any block, the probability is $R(\dot{q})$ that one or more elements in the block will participate in the loop, and the pins may be inserted in the loop with connections of acceptable length. If this reduces the yield too much, more sets of pins, or longer connections may be required.

5. Lattices Connected from Rectangular Arrays

Next we consider the connection of a square lattice from a rectangular array. Before proving a result on the connection distance necessary for this task, we present the following lemma concerning the separability of the square lattice.

Lemma 5: Consider the graph corresponding to a $K \times K$ square lattice. Any partition of the K^2 vertices into three sets A, B and C such that no vertex in A is connected to a vertex in B must satisfy

$$\min\left\{ ||A||,\ ||B|| \right\} \leq \frac{||C||^2 - ||C||}{2} .$$

The set C in such a partition is called a *cutset*.

In essence, the lemma bounds the size of the largest set which may be disconnected from the remaining (larger) part of the lattice by removing only a given number of vertices. It is readily apparent that choosing C, the set of removed vertices, to lie along a diagonal achieves the bound.

The proof of this lemma will be found in [15]. A slightly weaker version is proved in [16]. We now present the theorem.

Theorem 5: Consider an $N \times N$ square array with elements space unit distance apart. Let $K^2 = RN^2$. Then for and $0 < \delta < 1$ the probability that a $K \times K$ lattice can be connected tends to zero as $O(N^{\delta - 1} \log^2 N)$ unless the maximum connection distance satisfies

$$d \geq \sqrt{\frac{\delta R \log(N)}{-2(2+R)\log(p)}} = \Omega(\sqrt{\log N}). \qquad (5.1)$$

Proof: The proof proceeds in three steps First we define sets of array elements called *grids*. Then we show that there is, with probability approaching one, a grid with all its elements defective. Finally, we assume the existence of a defective grid and use Lemma 5 to show that if (5.1) is violated it is not possible to connect a lattice.

For any given N, K, and p, choose integer L such that

$$2+R \geq LR > 2. \tag{5.2}$$

For any $0<\delta<1$ let

$$m = \left\lfloor \sqrt{\frac{\delta \log(N)}{-L\log(p)}} \right\rfloor \qquad \text{and} \qquad M = L(m+1)^2. \tag{5.3}$$

($\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. Likewise, $\lceil x \rceil$ denotes the least integer greater than or equal to x).

Grids are subsets of the elements of the $N \times N$ array consisting of M columns of L blocks. Each block consists of $m \times m$ elements positioned as shown in Figure 5.1. All blocks are lined up in one of L horizontal bands. Each block column may take one of $\lfloor N/M \rfloor - m+1$ horizontal shift positions within fixed, non-overlapping regions.

This is made precise by the following definitions. Let e_{xy} be the element in row x and column y of the array. The j^{th} block column, $1 \leq j \leq M$, is a set

$$C_j = \bigcup_{l=0}^{L-1} \bigcup_{x=x_j}^{x_j+m-1} \bigcup_{y=l\lfloor N/L \rfloor+1}^{l\lfloor N/L \rfloor+m} \{e_{xy}\}$$

where x_j may be any integer

$$(j-1)\lfloor N/M \rfloor + 1 \leq x_j \leq j\lfloor N/M \rfloor + 1 - m. \tag{5.4}$$

Thus there are $\lfloor N/M \rfloor - m+1$ choices for each block column C_j.

A grid is a set

$$G = \bigcup_{j=1}^{M} C_j$$

for some choice of $x_1, x_2, \ldots, x_M$ satisfying (5.4). Thus there are $(\lfloor N/M \rfloor - m+1)^M$ grids.

The i^{th} gap in a band of a grid is the area between the pair of blocks in the $i-1^{st}$ and i^{th} block columns.

The width of the i^{th} gap is necessarily the same in any band and is given by

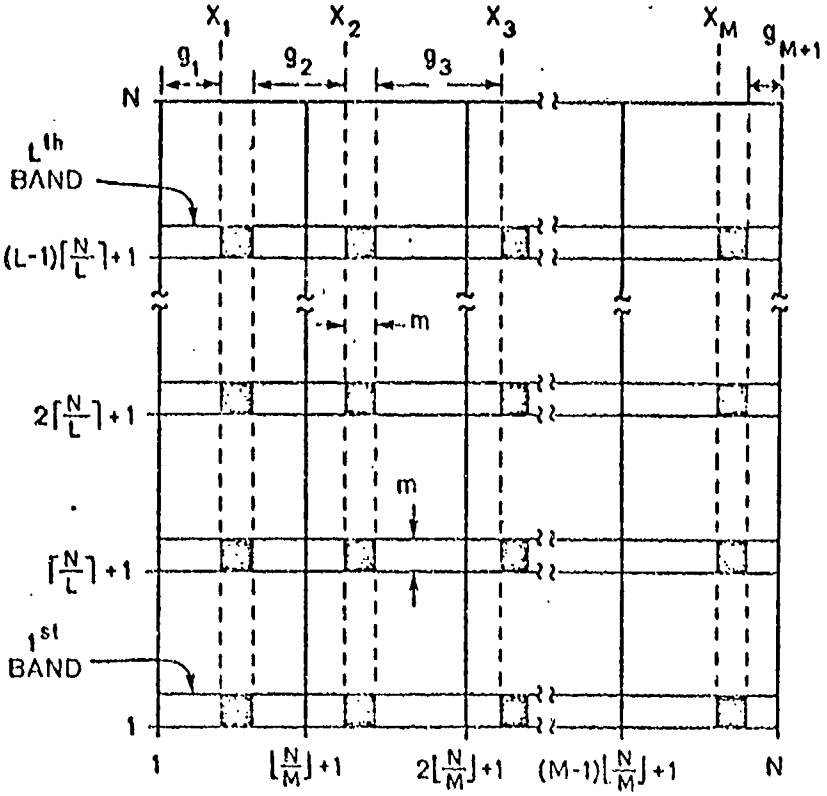

X_1
X_2
X_3
X_M
g_1
g_2
g_3
g_{M+1}
N
L^{th} BAND
$(L-1)\lceil \frac{N}{L} \rceil +1$
m
$2\lceil \frac{N}{L} \rceil +1$
m
$\lceil \frac{N}{L} \rceil +1$
1^{st} BAND
1
1
$\lfloor \frac{N}{M} \rfloor +1$
$2\lfloor \frac{N}{M} \rfloor +1$
$(M-1)\lfloor \frac{N}{M} \rfloor +1$
N

Fig. 5.1: The structure of a grid. The dark squares represent $m \times m$ blocks of defective elements.

$$g_i = \begin{cases} x_1 - 1 & i=1 \\ x_{i+1}-(x_i+m) & 2 \le i \le M \\ N+1-(x_N+m) & i=M+1 \end{cases} . \tag{5.5}$$

It is shown in Appendix 1 that with probability $1-O(N^{\delta-1}\log^4 N)$ there is a grid all elements of which are defective.

Suppose that the maximum connection distance d satisfies

$$d < (m+1)/\sqrt{2} \tag{5.6}$$

and that given an arbitrarily large $N \times N$ array with a completely defective grid, there is a way to connect a $K \times K$ lattice under constraint (5.6). We demonstrate a contradiction

Let V_0 be some subset of the K^2 connected elements which is located entirely between some pair of adjacent bands (or above the top band) and which is connected even after cutting all edges passing into the gaps. Then since V_0 lies between adjacent bands,

$$||V_0|| \leq N(\lceil N/L \rceil - m) < N^2/L. \tag{5.7}$$

Also let V_i and V_{M+1+i}, $1 \leq i \leq M+1$, be the set of elements connected to V_0 in or through the i^{th} gap of the upper and lower band, respectively, of the pair of bands surrounding V_0.

It is shown in Appendix 2 that we may always choose V_0 so that for all i

$$||V_0 \cup \bigcup_{j \neq i} V_j|| \geq ||V_i||. \tag{5.8}$$

Note that under assumption (5.6) on d, the connected lattice of elements cannot cross over or enclose a block of $m \times m$ elements which are all defective. (See Figure 5.2).

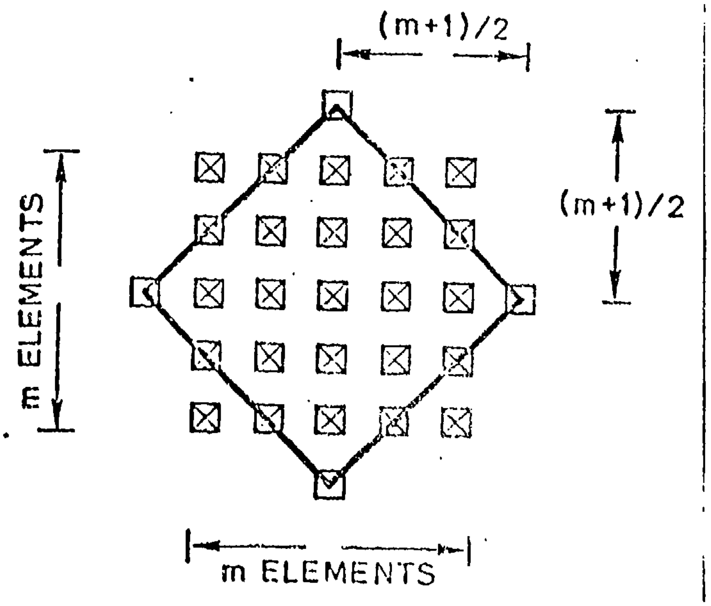


Fig. 5.2: The shortest connections enclosing an $m \times m$ block of defective elements. At least one connection must be of length $d \geq (m+1)/\sqrt{2}$.

Therefore any element connected to V_0 through one gap cannot be connected through a different gap as well or else a block would be enclosed. In other words, the sets V_i and V_j are disjoint for any $i \neq j$. A typical case is sketched in

Figure 5.3.

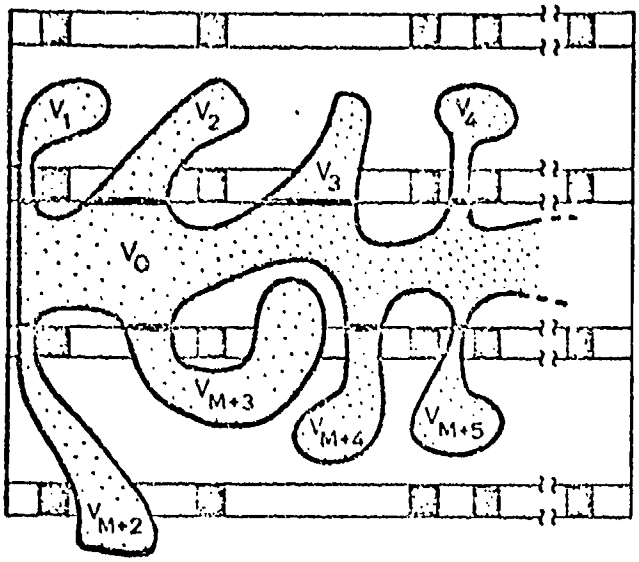


Fig. 5.3: A set of connected elements V_0 between two bands and the associated $\{V_i\}$. The dark squares represent blocks of defective elements.

Furthermore, if the gap through which V_i is connected has width g_i, there is a cutset of no more than $g_i(m+1)/\sqrt{2}$ elements whose removal disconnects V_i from the rest of the lattice. This is because V_i is connected only through the i^{th} gap, and since the given d is sufficient to penetrate only $(m+1)/\sqrt{2}$ layers of elements into the gap, by (5.6).

Noting that each connected element corresponds to a vertex in a $K \times K$ lattice, we use Lemma 5' to upper bound $||V_i||$ as follows. Identify $A = V_0 \cup \bigcup_{j \neq i} V_j$, C as the cutset, and $B = V_i \cap C^c$. By (5.8), $||B|| \leq ||A||$. We have

$$\begin{aligned} ||V_i|| &= ||B|| + ||C|| \\ &\leq \frac{||C||^2 + ||C||}{2} \qquad \text{by the lemma} \\ &\leq g_i^2(m+1)^2/4 + g_i(m+1)/2\sqrt{2} \qquad (5.9) \end{aligned}$$

as explained in the above paragraph.

From definitions (5.4) and (5.5), or a glance at Figure 5.3, we have

$$\begin{aligned}&0 \le g_j < 2N/M \qquad 1 \le j \le M+1\\ &\sum_{j=1}^{M+1} g_j < N.\end{aligned} \qquad (5.10)$$

Since there must be K^2 active elements in the lattice we have

$$\begin{aligned}
K^2 &= ||V_0|| + \sum_{i=1}^{2(M+1)} ||V_i|| \\
&< \frac{N^2}{L} + 2\sum_{j=1}^{M+1}\left[g_j^2(m+1)^2/4 + g_j(m+1)/2\sqrt{2}\right] && \text{by (5.7), (5.9)} \\
&= \frac{N^2}{L} + \frac{(m+1)^2}{2}\sum_{j=1}^{M+1} g_j^2 + \frac{(m+1)}{\sqrt{2}}\sum_{j=1}^{M+1} g_j \\
&< \frac{N^2}{L} + \frac{(m+1)^2}{2}\,\frac{M}{2}\left[\frac{2N}{M}\right]^2 + \frac{(m+1)}{\sqrt{2}}\,N && \text{under constraints (5.10)} \\
&< \frac{2N^2}{L} + O(N\sqrt{\log N}) && \text{by (5.3)} \\
&< K^2
\end{aligned}$$

for sufficiently large N by (5.2), which is a contradiction.

We conclude that the probability that the lattice can be connected goes to zero unless (5.6) is false, that is

$$\begin{aligned}
d &\ge (m+1)/\sqrt{2} \\
&\ge \sqrt{\frac{\delta R \log(N)}{-2(2+R)\log(p)}}
\end{aligned}$$

by definitions (5.2) and (5.3), which yields (5.1), completing the proof.▪

Unfortunately, Theorem 5 offers no lower bound on the number of wiring tracks t required between elements. We conjecture that $t = O(1)$ will suffice. If this is the case, it is easily shown that $d = O(\sqrt{\log N})$ is indeed all that is required. To demonstrate this, simply divide the $N \times N$ array into square blocks containing $c\log N$ elements for [illegible] constant c. If c is properly chosen, the Chernoff bound can be applied to show that for any $R < \bar{p}$ the probability that any particular block contains fewer than [illegible] active elements tends to zero exponentially in $\log N$. Since there are [illegible] $N^2/c\log N$ blocks, the probability that any of them has fewer than [illegible] elements also tends to zero. We therefore suppose that a $\sqrt{Rc\log N}\times\sqrt{Rc\log N}$ sublattice can be connected in each block with maximum connection distance $O(\sqrt{\log N})$. By providing $\sqrt{Rc\log N}$ tracks between adjacent blocks, which does not affect the order of growth of the total area, we can certainly connect the sublattices into one $\sqrt{R}N\times\sqrt{R}N$ lattice.

Leighton and Leiserson [7] have proposed a scheme which can connect any fraction $R<\bar{p}$ of the elements with probability $1-1/N$ using $d=O(\sqrt{\log N}\,\log\log N)$ and $t=O(\log\log N)$.

If the restriction to arrays of unit-square elements in Theorem 5 is relaxed, the following Corollary may be demonstrated.

Corollary: Consider and $N\times N$ rectangular array with elements of area A_e. Let $K^2=RN^2$. Then for any $0<\delta<1$, the probability that a $K\times K$ lattice can be connected tends to zero as $O(N^{\delta-1}\log^4 N)$ unless the maximum connection distance satisfies

$$d\geq\sqrt{\frac{A_e\,\delta R\log(N)}{-2(2+R)\log p}}=\Omega(\sqrt{A_e\log N}).$$

The proof is basically the same as that of Theorem 5, and will therefore be omitted. One difference is that the m-element by m-element defective blocks must be replaced by rectangular $m_h\times m_v$ blocks so that each block is still approximately square in terms of physical distance. The $\log^4 N$ term in the bound on the probability of connection, rather than the $\log^2 N$ term in Theorem 5, occurs because of the M^2 term in (A1.7). M must be proportional to the square of the number of elements on a vertical side of a defective block, m_v, which can now be $O(\log N)$ as compared with $m=O(\sqrt{\log N})$.

Thompson [17] has proposed a VLSI model in which the area of an element must increase linearly with the length of the longest wire it drives. This is justified by noting that the capacitance of a wire increases linearly with its length. The drive current needed to charge or discharge the wire in a given interval, and hence the area of the driving transistor, must therefore increase linearly with wire length. We therefore suppose that $A_t = \Omega(d)$. Substituting this into the result of the Corollary, we obtain d, $A_t = O(\log N)$.

These bounds may be achieved using selectors in the following way. We make the elements of width $O(1)$ and height $O(\log N)$, and arrange them in K rows of N elements. Between each pair of rows, we place two selectors, one connected to the upper row of elements and the other connected to the lower row. The selectors share a common row of K ports positioned between them. The lattice can be connected if a chain can be formed in each of the rows of elements and if every row can be connected to the ports above and below it by the adjacent selectors. The probability that this will be possible can easily be shown to approach one in light of the discussion of the linear array chain problem in Section 1 and by Theorem 2 with δ chosen less than 1/2.

Further work may be focussed on inventing a connection scheme with $d = O(\sqrt{\log N})$ and $t = O(1)$ or, alternatively, substantiating the assumption that the area of a driver must increase linearly with wire length.

Appendix 1

We show that a completely defective grid exists with probability tending to one. Let B be the number of grids G containing only defective elements. By the Chebyshev inequality,

$$P(B=0) \le P(|B-EB| \ge EB) \le \frac{Var\ B}{E^2\ B}. \tag{A1.1}$$

We upper bound the variance as follows:

$$\begin{aligned} Var\ B &= \sum_{\substack{G_1 \\ G_1 \cap G_2 \ne \phi}} \sum_{G_2} p^{||G_1 \cup G_2||} + \sum_{\substack{G_1 \\ G_1 \cap G_2 = \phi}} \sum_{G_2} p^{||G_1||+||G_2||} - \left[\sum_G p^{||G||} \right]^2 \\ &\le \sum_{\substack{G_1 \\ G_1 \cap G_2 \ne \phi}} \sum_{G_2} p^{||G_1 \cup G_2||} \\ &= p^{2LNm^2} \sum_{\substack{G_1 \\ G_1 \cap G_2 \ne \phi}} \sum_{G_2} p^{-||G_1 \cap G_2||}. \end{aligned}$$

Taking advantage of the restriction that G_2 intersects G_1 we assume the i^{th} block column is the first which overlaps, and expand the sum over G_2 into sums over the possible horizontal shifts of each block column in G_2. Let C_j and D_j denote the set of elements in the j^{th} block column of G_1 and G_2, respectively. Then

$$\begin{aligned} Var\ B \le p^{2LNm^2} \sum_{G_1} \sum_{i=1}^{M} \sum_{\substack{D_1: \\ C_1 \cap D_1 = \phi}} \cdots \sum_{\substack{D_{i-1}: \\ C_{i-1} \cap D_{i-1} = \phi}} \\ \sum_{\substack{D_i: \\ C_i \cap D_i \ne \phi}} p^{-||C_i \cap D_i||} \sum_{D_{i+1}} p^{-||C_{i+1} \cap D_{i+1}||} \cdots \sum_{D_M} p^{-||C_M \cap D_M||}. \end{aligned}$$

Note that the telescoping sums over the D_j's can be made a product of sums. This enables us to treat each sum separately, and upper bound them as follows. For the terms $1 \le j < i$,

$$\sum_{\substack{D_j: \\ C_j \cap D_j = \phi}} 1 \le \left\lfloor \frac{N}{M} \right\rfloor - m + 1 \tag{A1.2}$$

since there are this many horizontal shifts possible.

For the i^{th} term, letting k be the number of columns of *elements* in $G_i \cap D_i$,

$$\sum_{\substack{D_i:\\ G_i \cap D_i \neq \phi}} p^{-||G_i \cap D_i||} \leq p^{-Lm^2} + 2\sum_{k=1}^{m-1} p^{-Lmk}$$

$$= p^{-Lm^2}\left[1 + 2\,\frac{p^{Lm} - p^{Lm^2}}{1-p^{Lm}}\right]$$

$$< p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right] \qquad \text{(A1.3)}$$

since $m \geq 1$.

For the terms $i > j \geq M$, we can just add (A1.2) and (A1.3).

$$\sum_{D_j} p^{-||C_j \cap D_j||} < \left\lfloor \frac{N}{M} \right\rfloor - m + 1 + p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right]. \qquad \text{(A1.4)}$$

Bounding the terms $j \neq i$ by (A1.4) and the i^{th} term by (A1.3), and noting that there are $\left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1\right)^M$ possible grids G_1 and M possible first overlapping block columns i,

$$Var\ B < \left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1\right)^M p^{2LMm^2} M$$

$$\left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1 + p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right]\right)^{M-1} p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right]. \qquad \text{(A1.5)}$$

The expectation is

$$E\ B = \sum_G p^{||G||}$$

$$= \left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1\right)^M p^{LMm^2}. \qquad \text{(A1.6)}$$

Bringing together (A1.1), (A1.5) and (A1.6),

$$P(B=0) \leq \frac{Var\ B}{E^2\ B}$$

$$< \frac{M\left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1 + p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right]\right)^{M-1} p^{-Lm^2}\left[\frac{1+p^L}{1-p^L}\right]}{\left(\left\lfloor \frac{N}{M} \right\rfloor - m + 1\right)^M}$$

$$< M\left(1 + \frac{\left[\frac{1+p^L}{1-p^L}\right] N^\delta}{\frac{N}{M} - m}\right)^{M-1} \left[\frac{1+p^L}{1-p^L}\right] \frac{N^\delta}{\frac{N}{M} - m}$$

since $p^{-Lm^2} \le N^{\delta}$ from definition (5.3), and $\lfloor N/M \rfloor + 1 > N/M$. Making use of the relation $(1+x)^M \le \exp(Mx)$ to upperbound the first term, we have

$$P(B=0) < \exp\left(\frac{\left[\frac{1+p^L}{1-p^L}\right](M-1)\,M}{N^{1-\delta} - mMN^{-\delta}} \right) \left[\frac{1+p^L}{1-p^L}\right] \frac{M^2}{N^{1-\delta} - mMN^{-\delta}} \tag{A1.7}$$

$$= O(N^{\delta-1}\log^2 N)$$

since $M = O(\log N)$, $m = O(\sqrt{\log N})$ and L is a constant. So the probability that there is at least one completely defective grid approaches one.

Appendix 2

We wish to show that there is a V_0 such that all V_i constitute the smaller side of the partition induced by their respective defining gaps; that is

$$||V_0 \cup \bigcup_{j \ne i} V_j|| \ge ||V_i|| \qquad \text{for all } i. \tag{A2.1}$$

We begin by choosing some subset of the connected elements which is located entirely between some pair of adjacent bands (or above the top band) and which is connected even after cutting all connections passing into the gaps. Call this V_0, and let the corresponding $\{V_i\}$ be as defined following (5.7). Note that since there are K^2 connected elements, $||V_0|| + \sum ||V_i|| = K^2$. Thus there can be no more than one V_i with $||V_i|| > K^2/2$ and one of the following cases must apply.

Case 1: Suppose that the chosen V_0 is such that $||V_i|| \le K^2/2$ for all i. Then for all i, $||V_i|| \le K^2 - ||V_i|| = ||V_0 \cup \bigcup_{j \ne i} V_j||$, so that the chosen V_0 satisfies (A2.1).

Case 2: Suppose that there is exactly one V_i such that $||V_i|| > K^2/2$. Then we choose as a new V_0' the part of V_i which may be connected to V_0 by passing through only one gap and is located entirely between two bands. (See Figure A2.1). We repeat this procedure until Case 1 applies to the current V_0.

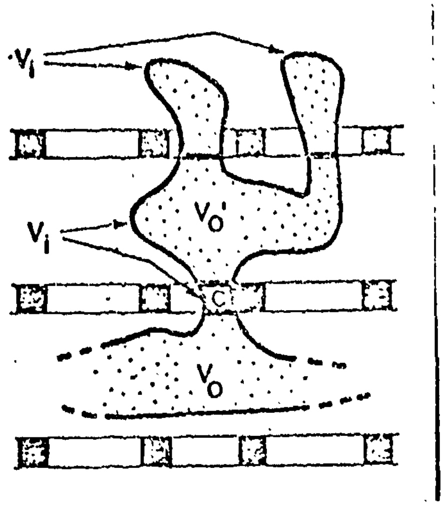


Fig. A2.1: The definition of a new V_0' from a given V_i. C is the set of connected elements in the gap, whose removal must separate V_0 from V_i.

We demonstrate that the procedure must terminate as follows. Let C be the set of elements in the gap. By the argument following (5.8), C is a cutset and $(||C||^2+||C||)/2$ is much less than $K^2/2$. By Lemma 5, either $||V_0 \cup \bigcup_{j\neq i} V_j|| \leq (||C||^2-||C||)/2$ or $||V_i|| \leq (||C||^2+||C||)/2$. (Note that $C \subset V_i$). Since in Case 2 $||V_i||>K^2/2$, it must be that

$$||V_0 \cup \bigcup_{j\neq i} V_j|| + ||C|| \leq (||C||^2+||C||)/2$$
$$< K^2/2.$$

Thus the procedure can never return through the gap from V_0' to V_0 and we can be assured that the procedure will never repeat a choice of V_0. Since there are only a finite number of possible V_0, the procedure must terminate, yielding a choice satisfying (A2.1).

References

1. Smith, R.T., et.al. Laser Programmable Redundancy and Yield Improvement in a 64K DRAM. IEEE J. Solid-State Circuits SC-16, 5 (Oct. 1981), pp. 506-513.

2. Lincoln Laboratories. Semiannual Technical Summary: Restructurable VLSI Program. ESD-TR-81-153 (March 31, 1981).

3. Hsia, Y., and Fedorak, R. Impact of MNOS/AWSI Technology on Reprogrammable Arrays. Symposium Record, Semi-Custom Integrated Circuit Technology Symposium May 26-27, 1981, Washington, DC.

4. Abbott, R., et. al. Equipping a Line of Memories with Spare Cells. Electronics (July 28, 1981) pp. 127-130.

5. Minato, O. et.al. HI-CMOS II 4K Static RAM. Digest of Technical Papers, 1981 IEEE Solid State Circuits Conf., pp.14-15.

6. Ziman, J.M. Models of Disorder. Cambridge Univ. Press, Cambridge, 1979.

7. Leighton, F.T., and Leiserson, C.E. Probabilistic Algorithms for Constructing Networks with Short Wires. Extended abstract available from authors.

8. Manning, F. An Approach to Highly Integrated Computer-maintained Cellular Arrays. IEEE Trans. Computers C-26, 6 (June 1977) pp. 536-552.

9. Aubusson, R., and Catt, I. Wafer-Scale Integration--A Fault Tolerant Procedure. IEEE J. Sol. State Circuits SC-13, 3 (June 1978) pp. 339-344.

10. Feller, W. An Introduction to Probability Theory and Its Applications. Vol. II. 2nd ed. Wiley, New York, 1971, pp. 194-198.

11. Broadbent, S.R., and Hammersley, J.M. Percolation Processes. I. Proc. Cambridge Phil. Soc. 53 (1957) pp. 629-641.

12. Frisch, H., Hammersley, J.M., and Welsh, D. Monte Carlo Estimates of Percolation Probabilities for Various Lattices. Physical Rev. 126,3 (May 1, 1962) pp. 949-951.

13. Harris, T. A Lower Bound for the Critical Probability in a Certain Percolation Process. Proc. Cambridge Phil. Soc. 56 (1960) pp. 13-20.

14. Fisher, M. Critical Probabilities for Cluster Size and Percolation Problems. J. Math. Phys. 2,4 (July-August 1961) pp. 620-627.

15. Greene, J.W. A Tight Non-Separability Proof for the Square Lattice. (In preparation).

16. Lipton, R.J., Eisenstat, S.C., and DeMillo, R.A. Space and Time Hierarchies for Classes of Control Structures and Data Structures. J. ACM 23,4 (Oct. 1976) pp. 726-727.

17. Thompson, C.D. A Complexity Theory for VLSI. (Dissertation). Carnegie-Mellon University (August 1980) pp. 41-44.

from Proceedings of MICRO 15

Optimizing Delayed Branches

Thomas R. Gross and John L. Hennessy

Departments of Electrical Engineering and Computer Science
Stanford University

Abstract

Delayed branches are commonly found in micro-architectures. A compiler or assembler can exploit delayed branches. This is achieved by moving code from one of several points to the positions following the branch instruction. We present several strategies for moving code to utilize the branch delay, and discuss the requirements and benefits of these strategies. An algorithm for processing branch delays has been implemented and we give empirical results. The performance data show that a reasonable percentage of these delays can be avoided.

Introduction

Recent research has focused on the relationship between compilers and computer architecture. The importance of computer architectures as hosts for compiled code is recognized as a dominant factor, and modern instruction sets are designed in close interaction with compiler writers. This development allows functionality to be provided either in hardware or in software[1].

Branch instructions are a major obstacle for pipelined machines. Most modern machines prefetch instructions before the preceding instructions have been completed. If one of the executing instructions was a branch instruction, the sequential successor of the branch instruction might not be the next instruction to be executed. If pipelining is also employed at the micro-instruction level, the micro-machine faces the same problems[2].

Conventional architectures employ additional hardware to cope with this problem. They detect the presence of a branch instruction and delay prefetching until the branch condition has been evaluated and the correct successor instruction has been obtained. Many pipelined machines with instruction lookahead use a branch prediction scheme to reduce the latency of obtaining the successor instruction. Most micro-architectures do only limited prefetching or do not support this feature in hardware at all[3]; they require that it be adhered to in software. This task becomes increasingly difficult in the presence of multi-way jump instructions[4].

Software techniques can be used to reduce the delay time associated with a branch. A software approach can also eliminate altogether the need for the hardware that detects the branch and prevents further execution until the successor is known. Since some number of sequential instructions following the branch are always executed, the branch optimizer must reorder instructions or insert no-ops to prevent the undesired execution of instructions. Both the RISC[5] and MIPS[6] architectures have no hardware branch delay so such a technique is needed. The IBM 801 uses a strategy that allows either hardware branch delay or execution of the sequential successor of the branch instruction[7].

There is an additional argument to consider in optimizing branch delays at the microprogram level. If microcode is used to directly support high level language features, special attention has to be given to branch instructions. These instructions are extremely frequent; benchmarks show that 25-30% of the instructions executed in some architectures are taken branches[8]. Branch delays are a dominant factor in machine speed and branch instructions are a major time consumer due to their adverse effect on pipelining[9]. Optimizers that eliminate the adverse effects of branch delay could be used to introduce branch instructions that are optionally delayed, or even branch instructions that always execute their sequential successor. The attractiveness of these choices depends on the effectiveness of optimization techniques like those presented in this paper.

The problem

In the following we will use the term *branch instruction* to refer to any instruction which changes the control flow. This includes conditional as well as unconditional branches and also trap instructions (which are called supervisor calls on some machines).

Delayed branches

In a pipelined machine, instruction *I* is fetched and started before some of its predecessors *I-1*, *I-2*, ... have completed. This strategy causes problems if there are data dependencies between these instructions[10]. In this paper we are concerned with the related problem, called *delayed branches*. Informally, a delayed branch of length *n* means that the *n* instructions following the branch are always executed whether or not the branch is taken.

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79 C-0680. Thomas Gross is supported by an IBM Graduate Fellowship.

Definition 1: Let instruction l be a branch instruction with target L. The branch is a *delayed branch* with delay n, if the sequence of instructions executed when the branch is taken is $l, l+1, \dots, l+n, L$.

The MIPS processor has instructions with branch delay 1 and 2. Indirect jumps have a branch delay of 2 as they involve a memory reference. All other control flow instructions (these are direct jumps, conditional branches, and trap instructions) have a branch delay of 1. The RISC processor[5] has branches with delay 1.

```
1              move    #1, r1
2              bequal  r1, r0, L12
3              sub     #1, r0
4              load    y, r2
5              add     r2, r0
6        L12:  store   r0, x
```

Figure 1: A simple example

Figure 1 gives an example. Assume that r0 has initially the value 1. If this code is executed on a machine with branch delay 1, the execution sequence is: 1-2-3-6 and the value 0 will be stored in location x.

In many environments, the responsibility to deal with this characteristic rests with individual programmer; instructions have to be moved manually during the coding of a program. This practice is error prone and not advisable if the instructions have different branch delays: it has to be automated if the instructions are generated by a compiler.

Software solutions

The simplest (completely unsatisfactory) approach for handling delayed branches is this : if the branch has delay n, then insert n no-ops immediately following the branch. But branches are very frequent in compiled code; padding each branch with n no-ops will result in excessive program size even for $n = 1$. The execution of large numbers of no-ops will severely degrade dynamic performance. Faced with only this choice, the hardware would be forced to implement branches so that only a time delay is incured and no extra instructions are executed. The alternative software approach is to move useful instructions after the branch instruction and resort to the insertion of no-ops only when no other instructions can be found. If this strategy is successful, the program may be significantly faster and only slightly larger.

There are two possible ways to handle code movement for delayed branches. The first approach puts the burden on the code generator; the second approach treats delayed branches in a post-pass after a standard code generation phase.

Several problems arise with the first approach. Code generators are normally fairly complex; adding this additional task makes code generation even more cumbersome. This strategy also requires that the output from the code generator is indeed the final and correct sequence of instructions. This assumption is not always possible or feasible. For example, it makes peephole optimization difficult, if not impossible. If there are other pipeline constraints or if some instruction packing is done, the code generator might not know the shape of the object program[10].

For these reasons we optimize delayed branches in a post-pass. This approach also allows us to optimize handwritten assembly language programs or microcode. The algorithms and optimizations in this paper can easily be adapted for use in a code generator that will be the final pass of a compiler.

Optimizing the delays

We now discuss the algorithm for the treatment of delayed branches. In this context we limit the discussion to conventional two-way branches. The extension to n-way jumps[4] is straightforward. We consider only resources visible at the level of assembly language, i.e. registers; however our model can be extended to include other resources as well.

Notations

Definition 2: Let instruction i be a branch instruction. We denote with ι^+ the branch target and with ι^- the location of the next instruction executed if the branch is not taken.

For example, if instruction i is *branch-on-equal R1, R2, L*, then $\iota^+ = L$ and $\iota^- = i+1$. Table 1 classifies different types of branch instructions according to our knowledge about the values of ι^+ and ι^- from the branch.

Group	ι^+ known	ι^- known
branch jump to subroutine jump direct	yes	not applicable
branch conditionally	yes	yes
trap SVC	no	yes
return to subroutine jump indirect	no	not applicable

Table 1: Control Flow Instructions

For trap instructions ι^+ is unknown, since although we know that execution resumes at a predefined location in the operating system after the trap has been raised, but we have no information about the instructions at this branch target.

The branch delay optimizer must also know the status of the relevant resources upon entry into a basic block. For each basic block B_j, $IN(B_j)$ is the set of registers which might be referenced in this block or any successor before they are written. The compiler provides this information; it is readily available in a compiler from the register allocation routines.

The information contained in $IN(B_j)$ must be conservatively correct in the dataflow sense: at entry into basic block j any register that could be read before it is written on any execution path starting with basic block j must be in the set $IN(B_j)$. The accuracy of the sets $IN(B_j)$ depends on how the register allocation process is done. If the compiler is using a global register allocation scheme, the information can be taken from the results of the data flow analysis. In the case of a simpler allocation scheme, e.g. allocating a variable to a register for the life of a procedure, the information can be easily estimated with reasonable accuracy. If the information has to be gathered from an ill-structured assembly program, the information will be pessimistic. Of course, IN can be arbitrarily enlarged to make gathering this information easier; however, large IN sets will limit the effectiveness of the optimizations.

Code motion

There are three major schemes for dealing with delayed branches. All three try to move useful instructions to the n positions after the branch instruction. They differ in the location from which they move the code and in the kind of improvement gained. These three schemes are:

1. Move n instructions from before the branch to after the branch.
2. Duplicate the first n instructions from t^+ and branch to the instruction at $t^+ + n$ instead.
3. Move the next n sequential instructions to immediately after the branch.

All movement of code is subject to the general requirement that there cannot be another branch instruction in the n instructions following a branch instruction. The additional requirements for each of the schemes are shown in Table 2.

The restrictions on the resources in $IN(B_j)$ for the basic block B_j which was not choosen guarantee that this optimization does not change the meaning of the program. Only resources which are "dead" outside of this basic block can be modified by optimizations 2 and 3.

The effect of optimization 1 is a simple movement of the branch instruction. No no-ops have to be inserted, and only useful instructions are executed, whether or not the branch is taken. This is an improvement in time and space over the default solution, the insertion of no-ops. In optimization scheme 2, the duplication of instructions does not reduce the size of the program over the no-op solution. However, execution of the code segment will be shortened by n cycles if the branch *is* taken. The third optimization will always reduce the size, and will reduce the execution time by n cycles whenever the branch ***is not*** taken.

Obviously, the first optimization is the most desirable. The relative advantages of optimizations 2 and 3 depend on the objectives of the optimization and on the dynamic properties of the program. If a branch instruction is taken k times and not taken l times, optimization 2 saves $n * k$ cycles. Optimization 3 will save $n * l$ cycles and n units of storage. Compilers which are mainly concerned with the size of the object program might favor the third optimization. If speed is important, optimization 2 is superior for all $k > l$. For loop-type branches, the relation $k >> l$ almost always holds; a large percentage of the branches executed in a program will be loop-type branches. The value of n is small (1 - 3) for current processor implementations; hence, the storage savings of optimization 3 will not be great.

Not all the optimizations are possible for all branch types as the requirements cannot always be fulfilled. Unconditional branches don't cause any problems in practice. Unconditional branches do not depend on the preceding instructions. Therefore the delay time can be fully utilized unless there are less than n instructions preceding the unconditional branch.

The branch instructions from groups 3 and 4 are the most problematic. For group 3 (the trap instructions) t^+ is unknown, so we may have to assume that $IN(t^+)$ is the set of all possible registers. Of course, most trap routines will use a standard entry

	Optimization	Requirements	Improvement
1	Move n instructions from before the branch till after the branch.	Always possible, but the branch cannot depend on the moved instructions.	Time Space
2	Duplicate the first n instructions from t^+ and branch to instruction $t^+ + n$ instead.	t^+ and $IN(t^-)$ must be known. The moved instructions cannot change any register R_i in $IN(t^-)$ or affect memory.	Time – only when branch is taken
3	Move the next n sequential instructions to right after the branch.	t^- and $IN(t^+)$ must be known. The moved instructions cannot change any register R_i in $IN(t^+)$ or affect memory.	Space Time – only when branch is not taken

Table 2: Requirements for optimization

sequence and the compiler could compute a smaller value of IN from that sequence. Likewise, the return instructions from group 4 usually employ a standard linkage convention, so that we can restrict the set IN(t^+) to some small subset. Furthermore, there are normally bookkeeping chores associated with procedure entry and exit. The stack pointer and/or the frame pointer might have to be recomputed and registers may have to be saved. These actions can be accomplished during the branch delay time in a well designed calling convention. In such cases, the branch optimizer should not attempt to reorder the code sequence.

The algorithm

The handling of delayed branches is done on a basic block basis but requires the use of the global information about future uses of the registers. We utilize basic blocks which have no jumps into them, if the basic block contains a jump, it must be the last instruction in the block.

Our algorithm recognizes the useful properties of unconditional branches and handles them accordingly. Other types of branch instructions are treated as follows: our algorithm begins by attempting optimization 1. If moving instructions from before the branch instruction to (the locations) directly after the branch instruction (optimization 1) moves n instructions, then no further consideration is necessary and the next basic block is processed. If this strategy moves only $k < n$ instructions then optimizations 2 and/or 3 are used to further improve the code.

In cases where optimizations 2 and 3 are both possible, attempts to use optimizations 2 and 3 must be ordered. The decision would be easier if we had some knowledge about the miss ratio of the branch, i.e. the number of times the branch is *not* taken over the number of times the branch is executed. There is an appealing heuristic that has been confirmed by numerous studies of program behavior' assume that backwards-going branches will be taken. Backwards-going branches are almost always loop branches (they always are in a structured program), and loop branches are almost always taken. Several studies have shown that forward-going branches are taken or not taken with almost equal probability[8]. Hence, optimization 3 should be preferred for forward-going branches because it saves both time and space.

Next we give the steps for processing a basic block. We assume that any other optimizations are complete and the shape of the block is determined.

1. Read in the basic block. If the basic block does not end with a branch instruction, leave the block unchanged and proceed to the next basic block. Otherwise determine n, the length of the branch delay, and obtain the set IN(B) for this basic block.
2. Try to move n instructions from before the branch instruction to after the branch instruction.
3. If step 2 moved only k with $k < n$ instructions, move n-k instructions from somewhere else:
    - If only one location, either t^+ or t^- is known, move instructions from this location.
    - If both t^+ and t^- are known, pick one of them and move instructions from there. Currently, if t^+ before the branch location (i.e. the branch is backward), t^+ is chosen. For forward branches, t^- is selected.

    Whenever instructions are moved, make sure that they don't effect memory or any register which is in IN(B_j) for the basic block B_j which was *not* chosen.
4. Modify the branch-target if step *3* moved k instructions from t^+ so that t^+ points to k instructions after the original target.

There is an additional reason for chosing scheme 2 for a backwards branch: the code at the branch target has already been analyzed and processed. Thus, instructions which can be moved are already available and only a single pass is needed.

Optimization cost

This algorithm checks each instruction of a basic block at most once during step 2. It starts with the instructions directly before the branch instruction and then goes on to the preceding instructions until n instructions are found or the basic block has been completely checked. Most of the moved instructions come from the end of a basic block; a practical improvement would restrict the search to the last l instructions. However, this is not done in the present implementation. Step 3 requires checking of additional instructions. Here at most n instructions will be considered per basic block. The search for movable instructions stops when an instruction violates the requirements in Table 2; any remaining slots are filled with no-ops.

In the worst case the time to process a basic block is proportional to the sum of its length and 2 * n. Since the number of basic blocks in a program is a linear function of the number of branches, the processing time for the entire program is O(Number of instructions + Branch count * n), or linear in the program length. This result compares favorably to the estimate of the resources needed to implement a hardware based approach[9].

Implementation

We have implemented a compiling system and optimizer for MIPS (Microprocessor without Interlocked Pipe Stages) an ongoing, experimental VLSI processor project[11]. Currently, compilers for Pascal, Fortran, and C exist. These compilers generate machine-language level instructions that ignore the effects of delayed branches. There is a branch optimizer that implements the technique described above; it also provides several other functions, such as limited instruction collapsing and compensation for resource interlocks.

MIPS Instructions

The original design for the MIPS architecture envisioned a branch delay of two for all control flow instructions. The high frequency of branch instructions in the output of the code generators motivated a redesign; now only those branches which involve a reference to memory have a branch delay of two. All other branches have a delay of one (see table 3).

Group	Delay
branch (pc relative)	1
jmp direct (absolute)	1
jmp via register	1
branch conditional	1
jmp indirect	2

Table 3: MIPS branch instructions

Effectiveness of the optimization

Tables 4 and 5 give results for some sample programs. They have been gathered by the branch optimizer and an instruction level simulator of MIPS. The optimizer also performs other functions, like instruction reordering and limited instruction packing. The effects of these operations are not considered here.

Space

Table 4 is a table of empirical results for static data. It shows the number of branches for each program and the percentage of no-ops that are removed after branches. For example, Puzzle I contains 124 branch instructions. With a branch delay of *n*, the 124 × *n* no-ops have to be inserted when no optimization is done. The optimization is able to reorder the code to use 52.4% of these instruction locations for a delayed branch of one and 47.6% of the locations for a branch delay of two. The test programs consist of

1. Fibonacci, a recursive implementation of computing a Fibonacci number.
2. Hanoi, an implementation of the "Towers of Hanoi" problem.
3. Queens, a program which positions 8 queens on a chess board.
4. Puzzle I, II, and III, three versions of the infamous Puzzle benchmark[12] that recursively solves a cube packing problem; the versions differ in their use of register variables (employed in versions II and III) and their treatment of arrays: versions I and II access array elements by indices, version III has been transformed to exploit advantages of the addressing mechanisms.

Time

```
 1  1      ld a, r6                 1      ld a, r6
 2  1      mov #0, r13              1      moy #0, r13
 3  1      ble r6, r13, L3          1      ble r6, r13, L3
 4  1      no-op                    1      sub #1, r12
 5 23  L1:sub #1, r12              23  L2:st r12, 3(sp)
 6 23      st r12, 3(sp)           23      st r13, 2(sp)
 7 23      st r13, 2(sp)           23      jsr foo
 8 23      jsr foo                 23      storepc 1(sp)
 9 23      storepc 1(sp)           23      st r0, n(r13)
10 23      st r0, n(r13)           23      add #1, r13
11 23      add #1, r13             23      bgt r6, r13, L2
12 23      bgt r6, r13, L1         23      sub #1, r12
13 23      no-op                    1  L3:....
14  1  L3:....
                  (a)                              (b)
```

Figure 2: Example for time saving

Figure 2 shows an example. All branch instructions have a branch delay of 1. The execution of the code segment shown in figure 2.a requires 211 cycles; this does not include time spent in the subroutine. Note that the procedure returns to the location following the storepc instruction.

The reordered instructions are shown in figure 2.b. As the first branch is forward, instructions from t^- are chosen. R12 cannot be in IN(B) for the basic block starting at label L3; this register will be decremented regardless of the outcome of the test in line 3. This optimization reduces the size of the object program. The next branch (in line 11 of figure 2.b) is backward and instructions from t^+ are selected. The first instruction from t^+ is duplicated and placed directly after the branch instruction. Note that the target of the branch has to be adjusted: the new target is the second instruction of the block, labeled here for clarity L2. The reordered version requires 188 cycles to execute, which is an improvement of approximately 10 %.

Program Name	Instruction count	Branch instructions	% Instructions sites used for delay 1	2	3	4	5
Fibonacci	44	10	90.0	65.0	55.5	45.0	35.0
Hanoi	52	7	85.7	50.0	35.0	32.1	28.5
Puzzle I	809	137	95.0	79.9	72.2	67.8	64.8
Puzzle II	580	137	76.9	56.5	48.9	41.4	36.2
Puzzle III	829	137	97.0	76.2	67.8	63.3	60.1
Queens	142	29	96.5	72.4	59.7	52.5	47.5

Table 4: Static improvements after branch optimization

Program Name	Branch delay utilized	Reduction in Space	Time
Fibonacci	76.0 %	13.7 %	10.2 %
Hanoi	66.6 %	9.8 %	7.7 %
Puzzle I	95.0 %	13.2 %	10.4.%
Puzzle II	78.2 %	18.0 %	16.6 %
Puzzle III	96.5 %	13.5 %	19.1 %
Queens	87.6 %	14.9 %	9.8 %

Table 5: Dynamic improvements after branch optimization

Compiler Interaction

Register Allocation

We assume in this paper that the register allocation is done outside the branch optimizer. Therefore, the final results obtainable in this two phase scheme may be less than optimal. The code generator heavily influences the availability of movable instructions. In Figure 3.a no instructions will be moved from before the branch to a location behind it. Figure 3.b shows a different register assignment that allows the first two instructions to be moved.

```
ld    x, r2              ld    x, r1
add   r2, r0             add   r1, r0
ld    y, r2              ld    y, r2
bzero r2, anywhere       bzero r2, anywhere

       (a)                      (b)
```

Figure 3: Register assignment

Unfortunately, the use of a large number of registers also has an adverse effect on code motion. As the set IN(B) becomes large, it inhibits the movement of instructions in step 3 of the algorithm.

Loops

There are two major schemes to translate high-level loop constructs into machine instructions shown in Figure 4. The first scheme uses a test at the beginning and branches to this test after execution of the loop body. The alternative approach tests the condition after each iteration at the end of the loop body. This scheme either tests the condition once at the start or branches to the test at the end of the loop to determine whether the loop is executed 0 times.

The choice has a high impact on the quality of the reordering. There are two reasons why the second scheme is superior to the first.

1. It executes one branch instruction less per iteration. In pipelined machines, this will result in substantially faster execution, especially for short loops.
2. No instructions from t^+ can be moved behind the branch in the first scheme if the test instruction is the first instruction at t^+. This restriction is a consequence of the general requirement that two branches be separated by the branch delay instructions.

One might argue that an unconditional branch at the end of a basic block does not incur any penalty. Unconditional branches have the nice property that the preceding instructions don't influence the branch. Step 2 of the outlined algorithm can therefore move n instructions from before the branch to a location behind it. But if the basic block is short, there will not be sufficient instructions to move.

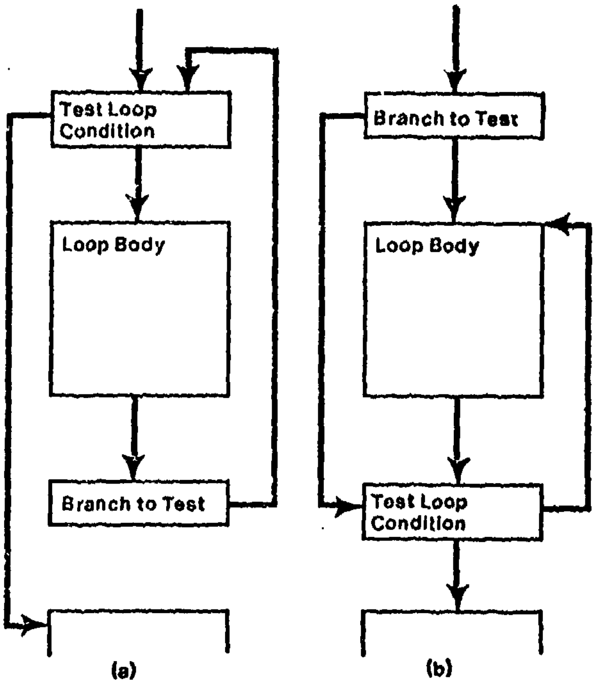


Figure 4: Loop constructs

The first version of the C compiler treated loops according to scheme 1 as shown in figure 4. A modified version translated loops differently and used the second scheme; it also eliminated branch chains. Table 6 shows the improvement gained from this change. Column A shows the speed-up over the first strategy: i.e. testing the loop at the bottom of a loop reduces the execution time of Fibonacci by 14.0%. Column B shows the improvements obtained by branch optimization for these programs when compared to the default solution, the insertion of no-ops. This column demonstrates the total speed-up obtainable.

Program Name	Reduction of execution time A	B
Fibonacci	14.0 %	22.7 %
Hanoi	6.2 %	13.6 %
Puzzle I	6.3 %	16.1 %
Puzzle II	9.3 %	24.3 %
Puzzle III	11.8 %	20.7 %
Queen	9.1 %	18.0 %

Table 6: Dynamic improvements after loop modification

Table 7 shows the contribution of the individual optimizations; it refers to the optimizations in Table 2. Each column gives the percentage of no-ops removed.

Name	% No-ops removed by			
	Opt1	Opt2	Opt3	Total
Fibonacci	50.0	0.0	25.0	75.0
Hanoi	25.0	0.0	50.0	75.0
Puzzle I	44.4	34.7	16.0	95.1
Puzzle II	48.6	15.2	22.8	86.8
Puzzle III	47.9	34.7	14.6	97.2
Queens	35.2	8.8	38.1	82.3

Table 7: Contribution of optimizations

Conclusion

Compiler technology can be used to enhance the design of computer instruction sets, to provide higher performance from those instruction sets, and to allow the implementation of schemes with substantially less hardware complexity. Branch delay elimination is an example of a technique that is becoming increasingly important in new architectures.

Acknowledgements

We thank John Gill and Jacobo Bulaevsky for the C compiler, Chris Rowen for the Pascal compiler. Norman Jouppi, Steven Przybylski, and the other members of the MIPS design team have provided useful suggestions during the course of this research.

References

1. Wulf, W.A., "Compilers and Computer Architecture," *Computer*, Vol. 14, No. 7, July 1981, pp. 41-48.

2. McClure, R.M., "Parallelism in Microprogrammed Controls," in *Intl. Advanced Summer Institute on Microprogramming*, Boulaye, G. and Mermet, J., eds., Hermann, Paris, 1972, pp. 307 - 328.

3. Agrawala, A.K. and Rauscher, T.G., *Foundations of Microprogramming*, Academic Press, New York, 1976, ACM Monograph Series

4. Fisher, J.A., "2^N-way Jump Microinstruction Hardware and an Effective Instruction Binding Method," *Proceedings: The 13th Annual Microprogramming Workshop Micro 13*, ACM, SIGMICRO, 1980, pp. 64 - 75.

5. Patterson, D.A. and Sequin C.H., "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. of the Eighth Annual Symposium on Computer Architecture*, Minneapolis, Minn., May 1981, .

6. Hennessy, J.,Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J., "MIPS: A Microprocessor Architecture," *Proceedings of Micro-15*, IEEE, October 1982, .

7. Radin, G., "The 801 Minicomputer," *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, , ACM, Palo Alto, March 1982, pp. 39 - 47.

8. Shustek, L.J., *Analysis and Performance of Computer Instruction Sets*, PhD dissertation, Stanford University, May 1977, Also published as SLAC Report 205.

9. Riseman, E.M. and Foster, C.C., "The Inhibition of Potential Parallelism by Conditional Jumps," *Trans. on Computer*, Vol. C-21, No. 12, Dec 1972, pp. 1405 - 1411.

10. Hennessy, J.L. and Gross, T.R., "Code Generation and Reorganization in the Presence of Pipeline Constraints," *Proc. Ninth POPL Conference*, ACM, January 1982, .

11. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill,J., "MIPS: A VLSI Processor Architecture," *Proc. CMU Conference on VLSI Systems and Computations*, Computer Science Press, October 1981, .

12. Baskett, F., "Puzzle: an informal compute bound benchmark", Widely circulated and run.

GRY: A PLA Minimizer
Programming Project
Lane Hemachandra

1 Purpose of the Project

This project implements a PLA minimizer using a novel *look-ahead* heuristic. The minimizer finds a minimal cover for an input PLA.

2 Problem Statement and Terminology

[Note: Until section 4, we will discuss the case of single-output functions.]

Consider a Boolean function f, in sum-of-products form. We can write f in *PLA format*, representing each product term as a line with its i^{th} element equal to

0 if the i^{th} literal appears in that term in complemented form
1 if the i^{th} literal appears in that term in uncomplemented form
2 if the i^{th} literal does not appear in that term

Example

$$f = x_1 x_3 \vee x_1 x_2 \bar{x}_3$$

In PLA format this becomes:

```
121
110
```

A *cube* is product term. We say a cube c is a *prime implicant* of a Boolean function f if it is a maximal cube of f. (For example, changing any 0 or 1 to a 2 in a prime implicant will result in a cube that has some intersection with $\bar{f}$.) A set of cubes is a ***cover*** (not to be confused with the verb "cover") for a function if the two are the same for every assignment of values to the literals. A cover C of f is ***minimal*** if it is an irredundant cover of prime implicants of f. By irredundant we mean that striking out any cube of C will leave a set of cubes that fail to cover some part of f. Given a function in PLA format (in a more general multi-output form discussed in section 4), these codes output a minimal cover for the PLA.

Example

$$f = \bar{a}b\bar{c} \vee \bar{a}bc \vee ab\bar{c}$$

In PLA format, this is:

```
010
011
110
```

A minimal cover for f is:

```
002
210
```

3 Architecture and Implementation

Note: Subroutine names appear in small boldface type.

These codes (Appendix 1), implemented in C on a VAX 11/780 at Stanford, are stored in Shasta:/vlsi/lah/PROGPROJ. The file OVERVIEW.GRY (Appendix 1) lists the subroutine hierarchy.

3.1 Method: Top Level main submain do_things

We first get a (possibly redundant) cover of prime implicants. Then we eliminate redundant cubes and thus have a minimal cover.

This final elimination (irredundant_cover) is straight-forward. We loop through the cubes and, for each cube, see if it is covered by the union of all the other cubes in the function. If it is, we immediately throw it out of our fanction. Clearly, the final output of this stage is still a cover of the function (since we only threw out cubes that were covered by what remained). Also, the output is certainly irredundant. Suppose that an output cube c was redundant. When we were looping through the cubes and got to c, all the cubes in the output were there (and possibly some more). Thus, at that time c was also redundant and would have been thrown out.

Now, let us discuss the important part of the algorithm: finding a cover of prime implicants (do_things). The crucial (though obvious) observation is:

> Observation Let j be a NDC (*non-don't-care*: a 0 or a 1) position of a cube c. Suppose that c raised in position j is not covered by f. Then for any cube d, $c \subseteq d$, d raised in position j is not covered by f.
>
> Proof Obvious. By assumption there is some minterm (a product term of all 0's and 1's) such that that minterm, m, complemented in position j is not in f. Since $c \subseteq d$, m is in d, so raising d in position j will give us the same non-f min-term as before. Thus, d raised in position j can not be covered by f.

Suppose we try and fail to raise a cube in some position j and then go on to successfully raise it in other positions. The observation above tells us that we still can not raise position j. Thus, our routine for the "raising to primality" stage is:

```
Mark all NDC positions in the PLA "active"
while (there are still active positions)
  beg
  choose an active position $p$ and mark it "not active"
  /* suppose that $p$ is the $j^{th}$ position of cube $i$ */
  if (cube $i$ complemented in position $j$ is covered by the PLA)
    raise position $j$ of cube $i$ to a 2
  end
```

Any ordering of the active positions will give us a cover of prime implicants. However, we would of course like an "intelligent" ordering that helps us contain and eliminate cubes as effectively as possible, even if this adds some bookkeeping work to our algorithm. Such a scheme is described in the next section.

In summary, the top-level flow is:

```
beg
  Initialize
  Get a cover of prime implicants
  Make the cover irredundant
end
```

3.2 The Look-Ahead Heuristic

Suppose we have a function. Recall that our goal is to contain as many cubes as possible. Thus, a reasonable way of choosing a position to raise might be to first try raising the position that is "nearest" to letting our cube cover other cubes. This is the idea behind the "look-ahead" heuristic. More exactly, at any time we raise the untried 0 or 1 that will let our cube contain as many cubes as possible if the raising succeeds. If more than one position is chosen by the previous rule, we try raising the one of these that will also bring us one "possible" raise away from covering as many more cubes as possible. If there is still more than one candidate, we break the tie by looking at distance three coverings, etc. By a "possible" raise, we simply mean a raise that we have not already tried and failed on.

Example

Consider the function

000
001
010
011
100

The most desirable position under the look-ahead heuristic would be the leftmost position of the fourth (counting 0 to 4) cube, since this is one away from covering one cube (cube zero) and two away from covering two cubes (cubes one and two) and three away from covering one cube (cube three). Tied for next best are the middle position of cube two and the rightmost position of cube one.

3.3 Implementation, Data Structures

3.3.1 The PLA

The array p contains the PLA, in PLA format (section 4). Henceforth, we assume that the PLA has H rows and W columns.

The Boolean "active positions" matrix implied in section 3.1 is a. $a[i][j]$ is true iff position j of cube i is active: it is a 0 or a 1 that we have not yet tried raising.

3.3.2 The Scoring Matrices

To choose the next active position under the look-ahead heuristic might intuitively seem to be very expensive. (A mindless implementation costs $O(H^2W^2)$ per choice for a total cost over all choices of $O(H^3W^3)$.) However, by keeping various partial results, and carefully updating them as the minimization proceeds, we can implement this more efficiently. (Essentially $O(HW)$ per choice. Total cost over all choices $O(H^2W^2)$. See section 3.4 for more details on the time cost.)

The crucial matrix that we will "use" is the Boolean (note: we assume 0 = "false" and 1 = "true") matrix b (for "bit-wise containments"). $b[i][j][k]$ is true iff cube k *fails* to cover cube i in position j. Though b was kept explicitly in early versions of this code, it has now been removed. A reference to b is implemented as a simple function, bits, that checks the appropriate two elements of p. This approach is a clear win. The time cost is the same (constant per reference to b). The space savings are dramatic! b was of size H^2W. With b eliminated, the biggest arrays that are left are of size max(HW,H^2). Thus, we can minimize much larger PLAs.

Let

$$r[i][j] = \sum_{k=0}^{W-1} b[i][k][j]$$

That is, $r[i][j]$ ("row sums") is the number of positions that keep cube j from covering cube i.

Let $f[i][j]$ be true iff either $i = j$ or cube j fails to cover cube i in some position where we have tried and failed to raise j. That is, $f[i][j]$ = true tells us it is impossible for cube j ever to cover cube i, except cubes are forbidden to cover themselves. $f[i][j]$ = false says nothing either way. f ("forget about possible containments") is of size HxH.

The "score" matrix will hold a value for each active position. The position with the largest entry in the score matrix will be the next position to be chosen by the look-ahead heuristic. The double precision score matrix is defined by (recall Booleans are 0-1):

$$s[i][j] = (a[i][j]) \sum_{k=0}^{H-1} \frac{(b[k][j][i])(1-f[k][i])}{(originalH+1)^{r[k][i]}}$$

Note that H is the current PLA size and $originalH$ was the original PLA size.

Crucially, we only use this nasty formula to initialize the score matrix. From then on, every time we do something (raise a position, fail to raise a position, contain a cube) that will effect the scoring matrices, we make whatever small changes are needed (section 3.3.3).

The score formula looks imposing, but it is really quite simple. The "$originalH+1$" simply imposes a radix system on the scoring, with the dominant weightings going to the "close" cubes of section 3.2.

Example
Continuing the example of 3.2, the score of the leftmost position of cube four is initially 1*(1/6) + 2*(1/36) + 1*(1/216) and the middle position of cube two and the rightmost position of cube one each score 1*(1/6) + 2*(1/36).

3.3.3 Implementation of the Look-Ahead Heuristic **do_failed do_no_cover do_we_cover compress**

3.3.3.1 Failed Raisings **do_failed**

Suppose we have just tried raising position $\langle i,j\rangle$ and have failed. Well, we may now suddenly know that cube i can never cover some other cubes (the ones that it now fails to cover in position j). Besides basic bookkeeping ($a[i][j]=0$; $s[i][j]=0$) this is

```
for each cube $k$ other that $i$
  if ($i$ does not cover $k$ in position $j$) and (we did not already know that $i$ could never cover $k$)
    beg
    note that $i$ can never cover $k$
    for each $l$ so that position $\langle i,l\rangle$ is active
      if $b[k][l][i]$ then $s[i][l] = s[i][l] - \frac{1}{(originalH+1)^{r[k][i]}}$
    end
```

3.3.3.2 Successful Raisings **do_raise**

There are two types of effects that successfully raising a position may have. First, things that used to cover our cube in the raised part may no longer do so. Secondly, the raised part may cover the corresponding parts of other cubes. Indeed we may even contain (section 3.3.3.3) other cubes.

Let us consider the effect of other cubes no longer covering our cube (**do_no_cover**). Suppose the position that we have just raised is $\langle y,x\rangle$. Then each cube which used to cover cube y in position x and no longer does may have its scores changed.

```
for each cube $q$ that used to cover cube y in position $x$ but no longer does
  beg
  increment the row sum $r[y][q]$ /* i.e. it fails in one more position */
  if we don't already know that it is impossible for cube $q$ to cover cube $y$ and $\langle q,x\rangle$ is not active
      beg
      note that it is impossible for cube $q$ to ever cover cube $y$
      for each column $j$ except $x$ if $b[y][j][q]$ then
      $s[q][j] = s[q][j] - \frac{1}{(originalH+1)^{r[y][q]-1}}$
    end
  else if we don't already know that it is impossible for cube $q$ to cover cube $y$ and $\langle q, x\rangle$ is active
    beg
    for each column $j$ except $x$ if $b[y][j][q]$
      $s[q][j] = s[q][j] + \frac{1}{(originalH+1)^{r[y][q]}} - \frac{1}{(originalH+1)^{r[y][q]-1}}$
    $s[q][x] = s[q][x] + \frac{1}{(originalH+1)^{r[y][q]}}$
    end
end
```

Finally, there is the effect of position $\langle y, x\rangle$ possibly covering new cubes (**do_we_cover**):

```
for each cube $n$ that cube $y$ used not to cover in position $x$ but now does
  beg
  decrement the row sum $r[n][y]$
  if we do not already know that it is impossible for cube $y$ to cover cube $n$
    beg
    /* remaining ones carry greater weight */
    for each column $j$ except $x$ if $b[n][j][y]$ and position $\langle y, j\rangle$ is active
      $s[y][j] = s[y][j] + \frac{1}{(originalH+1)^{r[n][y]}} - \frac{1}{(originalH+1)^{r[n][y]+1}}$
    end
  end
```

3.3.3.3 Contained Cubes compress

Finally, we come to the question of how to find and deal with contained cubes. Containment can only happen after a successful raising (say of position <y,x>). Any scores that have been boosted by not covering parts of a soon-to-be-destroyed cube must be decreased.

```
for $0 \leq d < H, d \neq py$ do
  if cube $d$ is contained by cube $y$ /* $r[d][y]$ is 0 */
    beg
    for cubes $i$ except $p$ and $d$ do
      if we do not already know it is impossible for cube $i$ to cover cube $d$
        for each position $j$ with ($b[d][j][i]$ and $\langle i, j\rangle$ active) do
          $s[i][j] = s[i][j] - \frac{1}{(originalH+1)^{r[d][i]}}$
    destroy cube $d$
    end
```

3.3.4 Implementation of Peripheral Functions covers tautology

3.3.4.1 Covers

Suppose we are given a cube c (with d twos) and a function f. We wish to know if the cube is covered (contained by) the function. Our approach will be to convert this question to a question of tautology. In particular, consider restricting all lines of f that intersect c to the don't care positions of c. (This is the restriction of f to c.) Simply ask if this restriction covers c. That is, is it a tautology (equal to $\{0,1\}^d$).

Example

$$c = 2120$$

$$\begin{aligned} f = &1110\\ &1201\\ &0220 \end{aligned}$$

Lines zero and three of f intersect c. Thus, c is covered by f iff the following is equivalent to 22 (it isn't):

11
02

3.3.4.2 Tautology

Suppose we are given a function f (of dimensionality d), and we want to see if it is a tautology. We begin by checking for various trivial cases that will give us a quick answer. Then we use a counting argument to try to get a quick answer. Count the number of minterms (0 dimensional cubes) contained in each cube of f (this is $2^{\text{number of twos in this cube}}$) and add these numbers. This gives an upper bound on the number of minterms in f (the figure is exact when the cubes of f are disjoint). We can immediately conclude that f is not a tautology if the upper bound is less than the the size of $\{0,1\}^d$ (which is 2^d). For example
02
11
has a count of three so it certainly can not cover 22 which has four minterms.

If these techniques do not give us an answer, we want to reduce this d-dimensional tautology question to two $(d-1)$-dimensional tautology questions. There are a number of heuristics that we could use to choose which of the d-dimensions we will split on. One way would be to try having almost equal number of cubes in each of the subproblems. We use a slightly more interesting method. Split on the dimension (say column j) which makes the sum of the number of cubes in the two subproblems minimum (that is, the column with the fewest twos). Split by putting all cubes with a 0 or a 2 in column j into one subproblem and cubes with a 1 or 2 in column j into the second subproblem (and eliminate column j in both subproblems). A very useful thing that we do is to solve the smaller subproblem first. If this subproblem is not a tautology, then we can return the fact that f is not a tautology without even solving the larger subproblem. Otherwise, return the solution to the larger subproblem.

Example

$$\begin{aligned} f = &012\\ &201\\ &112 \end{aligned}$$

We split on the middle column and ask the tautology question for 21 which fails by vertex counting. Thus f is not a tautology.

3.3.5 Error Checking

The program contains extensive debugging and internal consistency checks. These come in three flavors, which are turned on and off by a switch line in the input file.

The variable *dbg* controls the printing of extensive debugging information that shows the flow and actions of the program. This switch should be left off by the average user.

More interesting is *dbgerr*. This turns on checks within the program for impossible situations. If one occurs, the routine panic is called to tell the user what error has been encountered and given him the option of bailing out. The checks turned on be *dbgerr* are not expensive, so it should be left on by most users. Needless to say, the amusing message printed by panic should never be seen by any user.

Finally, the variable *IRS* turns on some very extensive error checking: after each attempted raise, the values for all the scoring matrices (except *a*) are recomputed from the PLA and compared with the values currently in the matrices (audit). This assures us that the complex score adjustment algorithms are correct. This was extremely useful in debugging the program, as errors were spotted as soon as they were introduced. However, this check is *very* expensive and should be left off by most users.

The exact method of specifying these variables is described in section 4.

3.4 Time Costs

3.4.1 Look-Ahead Time Costs

It is immediate from the algorithms in 3.3.3.1 and 3.3.3.2 that a successful or failed raising costs $O(HW)$ to choose the part and maintain the scoring matrices, excluding the cost for containing cubes. At most HW positions are checked for raising, so the total cost is $O(H^2W^2)$. The cost of containing k cubes is $O(kHW)$. Each cube can only be contained once during the course of the algorithm, so the total cost of containment adjustments is $O(H^2W)$. Thus, the total cost over the run of the algorithm of the look-ahead heuristic is $O(H^2W^2)$.

3.4.2 Covering Time Costs

The covering/tautology costs are trickier. The worst case costs of our implementation are exponential in the number of input columns of the PLA (the non-tautology question is NP-complete). However, in practise the covering question can usually be quickly answered. When the answer is no, all we have to do is find some part of the off-set (or reduce to some problem that vertex counting can answer). When the answer is yes, our search usually cuts off quickly since we can quit with a line of twos (and our splitting heuristic tries to preserve twos and split away ones and zeros).

4 Multiple Outputs / Input Format: Using the Code

In real PLAs, we also have an output plane. This corresponds to having a number of Boolean functions, which we allow to share product terms. We simply set the i^{th} element of the output plane of cube j to 1 if the product term j appears in function i, and we set it to 0 otherwise.

Example

$$f_1 = x_1x_3 \vee x_1x_2\bar{x}_3$$

$$f_2 = x_1x_2 \vee x_1x_2\bar{x}_3$$

$$f_3 = x_1x_2 \vee x_1x_3$$

in PLA format this becomes:
121 101
110 110
112 011

The look-ahead heuristic is unchanged, except we must remember that, in the output parts, 1 covers anything and 0 covers 0 but not 1. The covering and tautology extensions are straightforward. In covering, we restrict our tautology check (and our selection of intersecting cubes) to the output columns that are on in the given cube. In tautology we just modify the basis cases. (If some output component is included in no cube of the function, we can quit right away, etc.)

The input to the function is a file containing a PLA in the multi-output format described above, starting on its third line. The first line should contain three integers. First, the number of rows in the PLA. Second, the number of input columns. And finally, the number of output columns. The second line should contain, in order, the debugging switches *dbg*, *dbgerr*, and *IRS* of section 3.3.5. A 1 means the variable is on (true) and a zero means it is off (false). The suggested second line is 0 1 0. Thus a simple input file might look like:

```
4 3 2
0 1 0
110 10
112 11
121 11
010 01
```

Blanks within lines are ignored. The output file is of exactly the same format as the input file!

To run the minimizer, simply type "gry f1 f2" in the directory where the codes reside (SHASTA: /vlsi/lah/PROGPROJ). f1 is the input file and f2 is the name of the output file. (Defaults are "input.pla" and "output.pla".)

5 Test Run Descriptions

Appendix 2 contains script files of program sessions. There is a set of "simple" runs that minimize trivial PLAs, and a set of "sample" runs that minimize less trivial PLAs. Run times are included with each run. The profiled runs "auditoff" and "auditon" show how our implementation of the look-ahead heuristic (it takes 40% of the runtime) compares with the straight-forward implementation (where it takes 85% of the much longer runtime).

6 Comments and Conclusions

As it is extremely expensive to find *minimum* PLAs, we are motivated to find "good" PLAs. These codes guarantee to find a *minimal* cover for any input PLA, and use a powerful look-ahead heuristic to try to find a good minimal cover. The space cost is low, $O(\max(H^2, HW))$. The time cost is low enough to allow minimization of very large PLAs.

7 Citations and Acknowledgments

This project was written during my research assistantship in Professor Ullman's VLSI group.

The only previous treatment of a look-ahead heuristic for PLA minimization that I know of is *A Comparison of Logic Minimization Strategies Using ESPRESSO: An APL Program Package for Partitioned Logic Minimalization*, by Brayton, Hachtel, Hemachandra, Newton, and Sangiovanni-Vincentelli, to appear in the Proc. of IEEE Intl. Conf. on Circuits and Computers, May, 1982, Rome. This paper discusses an implementation of the look-ahead heuristic in a "local" sense. That is, we choose some ordering of the cubes in our function, and then go through the cubes in this order. Each cube is raised to a prime implicant by using the look-ahead heuristic to choose the ordering of its parts. We also mentioned that implementing a "global" look-ahead heuristic was an area in which future effort should be focused.

from Proceedings of MICRO 15

MIPS: A Microprocessor Architecture

John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen,
Thomas Gross, Forest Baskett, and John Gill

Departments of Electrical Engineering and Computer Science
Stanford University

Abstract

MIPS is a new single chip VLSI microprocessor. It attempts to achieve high performance with the use of a simplified instruction set, similar to those found in microengines. The processor is a fast pipelined engine without pipeline interlocks. Software solutions to several traditional hardware problems, such as providing pipeline interlocks, are used.

Introduction

MIPS (Microprocessor without Interlocked Pipe Stages) is a new general purpose microprocessor architecture designed to be implemented on a single VLSI chip. The main goal of the design is high performance in the execution of compiled code. The architecture is experimental since it is a radical break with the trend of modern computer architectures. The basic philosophy of MIPS is to present an instruction set that is a compiler-driven encoding of the microengine. Thus, little or no decoding is needed and the instructions correspond closely to microcode instructions. The processor is pipelined but provides no pipeline interlock hardware; this function must be provided by software.

The MIPS architecture presents the user with a fast machine with a simple instruction set. This approach has been used by the IBM 801 project[1] and is currently being explored by the RISC project at Berkeley[2]; it is directly opposed to the approach taken by architectures such as the VAX. However, there are significant differences between the RISC approach and the approach used in MIPS:

1. The RISC architecture is simple both in the instruction set and the hardware needed to implement that instruction set. Although the MIPS instruction set has a simple hardware implementation (i.e. it requires a minimal amount of hardware control), the user level instruction set is not as straightforward, and the simplicity of the user level instruction set is secondary to the performance goals.

2. The thrust of the RISC design is towards efficient implementation of a straightforward instruction set. In the MIPS design, high performance from the hardware engine is a primary goal, and the microengine is presented to the end user with a minimal amount of interpretation. This makes most of the microengine's parallelism available at the instruction set level.

3. The RISC project relies on a straightforward instruction set and straightforward compiler technology. MIPS will require more sophisticated compiler technology and will gain significant performance benefits from that technology. The compiler technology allows a microcode-level instruction set to appear like a normal instruction set to both code generators and assembly language programmers.

The MIPS architecture is closer to the 801 architecture in many aspects. In both machines the macroinstruction set maps very directly to the microoperations of the processor. Both processors may be thought of as architectures with micro-level user instruction sets. Microcode is created by compilers and code generators as it is needed to implement complex operations. The primary differences lie in various architectural choices about pipeline design, registers, opcodes and in the attempt in the MIPS instruction set to make all the microengine parallelism available at the user instruction set level. These attempts are most visible within MIPS in the following ways: the two-part memory/ALU and ALU/ALU instructions, the explicit pipeline interlocks, and the conditional jump instructions.

MIPS is designed for high performance. To allow the user to get maximum performance, the complexity of individual instructions is minimized. This allows the execution of these instructions at significantly higher speeds. To take advantage of simpler hardware and an instruction set that easily maps to the microinstruction set, additional compiler-type translation is needed. This compiler technology makes a compact and time-efficient mapping between higher level constructs and the simplified instruction set. The shifting of the complexity from the hardware to the software has several major advantages:

- The complexity is paid for only once during compilation. When a user runs his program on a complex architecture, he pays the cost of the architectural overhead each time he runs his program.
- It allows the concentration of energies on the software, rather than constructing a complex hardware engine, which is hard to design, debug, and efficiently utilize. Software is not necessarily easier to construct, but the VLSI environment makes hardware simplicity important.

The design of a high performance VLSI processor is dramatically affected by the technology. Among the most important design considerations are: the effect of pin limitations, available silicon

0194-1895/82/0000/0017$00.75 © 1982 IEEE

area, and size/speed tradeoffs. Pin limitations force the careful design of a scheme for multiplexing the available pins, especially when data and instruction fetches are overlapped. Area limitations and the speed of off-chip intercommunication require choices between on- and off-chip functions as well as limiting the complete on-chip design. With current state-of-the-art technology either some vital component of the processor (such as memory management) must be off-chip, or the size of the chip will make both its performance and yields unacceptably low. Choosing what functions are migrated off-chip must be done carefully so that the performance effects of the partitioning are minimized. In some cases, through careful design, the effects may be eliminated at some extra cost for high speed off-chip functions.

Speed/complexity/area tradeoffs are perhaps the most important and difficult phenomena to deal with. Additional on-chip functionality requires more area, which also slows down the performance of every other function. This occurs for two equally important reasons: additional control and decoding logic increases the length of the critical path (by increasing the number of active elements in the path) and each additional function increases the length of internal wire delays. In the processor's data path these wire delays can be substantial, since thy accumulate both from bus delays, which occur when the data path is lengthed, and control delays, which occur when the decoding and control is expanded or when the data path is widened. In the MIPS architecture we have attempted to control these delays; however, they remain a dominant factor in determining the speed of the processor.

The microarchitecture

Design philosophy

The fastest execution of a task on a microengine would be one in which all resources of the microengine were used at a 100% duty cycle performing a nonredundant and algorithmically efficient encoding of the task. The MIPS microengine attempts to achieve this goal. The user instruction set is an encoding of the microengine that makes a maximum amount of the microengine available. This goal motivated many of the design decisions found in the architecture.

MIPS is a load/store architecture, i.e. data may be operated on only when it is in a register and only load/store instructions access memory. If data operands are used repeatedly in a basic block of code, having them in registers will prevent redundant load/stores and redundant addressing calculations; this allows higher throughput since more operations directly related to the computation can be performed. The only addressing modes supported are immediate, based with offset, indexed, or base shifted. These addressing modes may require fields from the instruction itself, general registers, and one ALU or shifter operation. Another ALU operation available in the fourth stage of every instruction can be used for a (possibly unrelated) computation. Another major benefit derived from the load/store architecture is simplicity of the pipeline structure. The simplified structure has a fixed number of pipestages, each of the same length. Because, the stages can be used in varying (but related) ways, pipline utilization improves. Also, the absence of synchronization between stages of the pipe, increases the performance of the pipeline and simplifies the hardware. The simplified pipeline eases the handling of both interrupts and page faults.

Although MIPS is a pipelined processor it does not have hardware pipeline interlocks. This approach is often seen in low and medium performance microengines. MIPS five stage pipeline contains three active instructions at any time; either the odd or even pipestages are active. The major pipestages and their tasks are shown in Table 1.

Table 1: Major pipestages and their functions

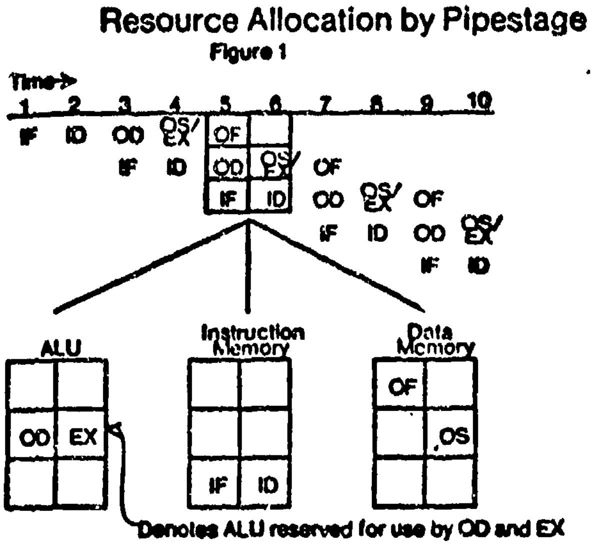
Stage	Mnemonic	Task
Instruction Fetch	IF	Send out the PC, increment it
Instruction Decode	ID	Decode instruction
Operand Decode	OD	Compute effective address and send to memory if load or store, use ALU
Operand Store/ Execution	OS/ EX	Store: write operand/ Execution: use ALU
Operand Fetch	OF	Load: read operand

Interlocks that are required because of dependencies brought out by pipelining are *not* provided by the hardware. Instead, these interlocks must be statically provided where they are needed by a *pipeline reorganizer*. This has two benefits:

1. A more regular and faster hardware implementation is possible since it does not have the usual complexity associated with a pipelined machine. Hardware interlocks cause small delays for all instructions, regardless of their relationship on other instructions. Also, interlock hardware tends to be very complex and nonregular[3, 4]. The lack of such hardware is especially important for VLSI implementations, where regularity and simplicity is important.
2. Rearranging operations at compile time is better than delaying them at run time. With a good pipeline reorganizer, most cases where interlocks are avoidable should be found and taken advantage of. This results in performance better than a comparable machine with hardware interlocks, since usage of resources will not be delayed. In cases where this is not detected or is not possible, no-ops must be inserted into the code. This does not slow down execution compared to a similar machine with hardware interlocks, but does increase code size. The shifting of work to a reorganizer would be a disadvantage if it took excessive amounts of computation. It appears this is not a problem for our first reorganizer.

In the MIPS pipeline resource usage is permanently allocated to various pipe stages. Rather than having pipeline stages compete for the use of resources through queues or priority schemes, the machine's resources are dedicated to specific stages so that they are 100% utilized. In Figure 1, the allocation of resources to individual pipe stages is shown. When concurrently executing pipe stages are overlayed, all available resources can be used.

Figure 1: Resource Allocation by Pipestage



To achieve 100% utilization primitive operations in the microengine (e.g., load/store, ALU operations) must be completely packed into macroinstructions. This is not possible for three reasons:

1. Dependencies can prevent full usage of the microengine, for example when a sequence of register loads must be done before an ALU operation or when no-ops must be inserted.
2. An encoding that preserved all the parallelism (i.e., the microcontrol word itself) would be too large. This is not serious problem since many of the possible microinstructions are not useful.
3. The encoding of the microengine presented in the instruction set sacrifices some functional specification for immediate data. In the worst case, space in the instruction word used for loading large immediate values takes up the space normally used for a base register, displacement, and ALU operation specification. In this case the memory interface and ALU can not be used during the pipe stage for which they are dedicated.

Nevertheless, first results on microengine utilization are encouraging. Many instructions fully utilize the major resources of the machine. Other instructions, such as load immediate which use few of the resources of the machine, would mandate greatly increased control complexity if overlap with surrounding instructions was attempted in an irregular fashion.

MIPS has one instruction size, and all instructions execute in the same amount of time (one data memory cycle). This choice simplifies the construction of code generators for the architecture (by eliminating many nonobvious code sequences for different functions) and makes the construction of a synchronous regular pipeline much easier. Additionally, the fact that each macroinstruction is a single microinstruction of fixed length and execution time means that a minimum amount of internal state is needed in the processor. The absence of this internal state leads to a faster processor and minimizes the difficulty of supporting interrupts and page faults.

Resources of the microengine

The major functional components of the microengine include:

- ALU resources: A high speed, 32-bit carry lookahead ALU with hardware support for multiply and divide; and a barrel shifter with byte insert and extract capabilities. Only one of the ALU resources is usable at a time. Thus within the class of ALU resources, functional units can not be fully used even when the class itself is used 100%.
- Internal bus resources: Two 32-bit bidirectional busses, each connecting almost all functional components.
- On chip storage: Sixteen 32-bit general purpose registers.
- Memory resources: Two memory interfaces, one for instructions and one for data. Each of the parts of the memory resource can be 100% utilized (subject to packing and instruction space usage) because either one store or load form data memory and one instruction fetch can occur simultaneously.
- A multistage PC unit: An incrementable current PC with storage of one branch target as well as four previous PC values. These are required by the pipelining of instructions and intercupt and exception handling.

The Instruction set

All MIPS instructions are 32-bits. The user instruction set is a compiler-based encoding of the micromachine. Static and dynamic instruction set efficiency, as determined by a code generator, is used to decide what micromachine features to encode into macroinstructions in the architecture. Multiple simple (and possibly unrelated) instruction pieces are packed together into an instruction word. The basic instruction pieces are:

1. ALU pieces - these instructions are all register/register (2 and 3 operand formats). They all use less than 1/2 of an instruction word. Included in this category are byte insert/extract, two bit Booths multiply step, and one bit nonrestoring divide step, as well as standard ALU and logical operations.
2. Load/store pieces - these instructions load and store

memory operands. They use between 16 and 32 bits of an instruction word. When a load instruction is less than 32 bits, it may be packaged with an ALU instruction, which is executed during the Execution stage of the pipeline.

3. Control flow pieces - these include direct jumps and compare instructions with relative jumps. MIPS does not have condition codes, but includes a rich collection of set conditionally and compare and jump instructions. The set conditional instructions provide a powerful implementation for conditional expressions. They set a register to all 1's or 0's based on one of 16 possible comparisons done during the operand decode stage. During the Execution stage an ALU operation is available for logical operations with other booleans. The compare and jump instructions are direct encodings of the micromachine: the operand decode stage computes the address of the branch target and the Execution cycle does the comparison. All branch instructions have a delay in their effect of one instruction; i.e., the next sequential instruction is always executed.
4. Other instructions - include procedure and interrupt linkage. The procedure linkage instructions also fit easily into the micromachine format of effective address calculation and register-register computation instructions.

MIPS is a word-addressed machine. This provides several major performance advantages over a byte addressed architecture. First, the use of word addressing simplifies the memory interface since extraction and insertion hardware is not needed. This is particularly important, since instruction and data fetch/store are in a critical path. Second, when byte data (characters) can be handled in word blocks, the computation is much more efficient. Last, the effectiveness of short offsets from base register is multiplied by a factor of four.

MIPS does not directly support floating point arithmetic. For applications where such computations are infrequent, floating point operations implemented with integer operations and field insertion/extraction sequences should be sufficient. For more intensive applications a numeric co-processor similar to the Intel 8087 would be appropriate.

Systems issues

The key systems issues are the memory system, and internal traps and external interrupt support.

The memory system

The use of memory mapping hardware (off chip in the current design) is needed to support virtual memory. Modern microprocessors (Motorola 68000) are already faced with the problem that the sum of the memory access time and the memory mapping time is too long to allow the processor to run at full speed. This problem is compounded in MIPS; the effect of pipelining is that a single instruction/data memory must provide access at approximately twice the normal rate (for 64k RAMS).

The solution we have chosen to this problem is to separate the data and instruction memory systems. Separation of program and data is a regular practice on many machines; in the MIPS system it allows us to significantly increase performance. Another benefit of the separation is that it allows the use of a cache only for instructions. Because the instruction memory can be treated as read-only memory (except when a program is being loaded), the cache control is simple. The use of an instruction cache allows increased performance by providing more time during the critical instruction decode pipe stage.

Faults and Interrupts

The MIPS architecture will support page faults, externally generated interrupts, and internally generated traps (arithmetic overflow). The necessary hardware to handle such things in a pipelined architecture usually large and complex[3,4]. Furthermore, this is an area where the lack of sufficient hardware support makes the construction of systems software impossible. However, because the MIPS instruction set is not interpreted by a microengine (with its own state), hardware support for page faults and interrupts is significantly simplified.

To handle interrupts and page faults correctly, two important properties are required. First, the architecture must ensure correct shutdown of the pipe, without executing any faulted instructions (such as the instruction which page faulted). Most present microprocessors can not perform this function correctly (e.g. Motorola 68000, Zilog Z8000, and the Intel 8086). Second, the processor must be able to correctly restore the pipe and continue execution as if the interrupt or fault had not occurred.

These problems are significantly eased in MIPS because of the location of writes within the pipe stages. In MIPS all instructions which can page fault do not write to any storage, either registers or memory, before the fault is detected. The occurrence of a page fault need only turn off writes generated by this and any instructions following it which are already in the pipe. These following instructions also have not written to any storage before the fault occurs. The instruction preceding the faulting instruction is guaranteed to be executable or to fault in a restartable manner even after the instruction following it faults. The pipeline is drained and control is transferred to a general purpose exception handler. To correctly restart execution three instructions need to be reexecuted. A multistage PC tracks these instructions and aids in correctly executing them.

Software issues

The two major components of the MIPS software system are compilers and *pipeline reorganizers*. The input to a pipeline reorganizer is a sequence of simple MIPS instructions or instruction pieces generated without taking the pipeline interlocks and instruction packing features into account. This relieves the compiler from the task of dealing with the restrictions that are imposed by the pipeline constraints on legal code sequences. The reorganizer reorders the instructions to make maximum use of the pipeline while enforcing the pipeline interlocks in the code. It also packs the instruction pieces to maximize use of each instruction word. Lastly, the pipeline reorganizer handles the effect of branch delays. This software is an important part of the MIPS architecture. It is responsible for making the low-level microarchitecture into a usable and comprehensible instruction set. Since the exact details of pipeline interlocks and branch delays may change between implementations, the architecture is actually defined by the input to the pipeline reorganizer.

Since all instructions execute in the same time, and most instructions generated by a code generator will not be full MIPS instruction set, the instruction packing can be very effective in reducing execution time. In fully packed instructions, e.g. a load combined with an ALU instruction, all the major processor resources (both memory interfaces, the alu, busses and control logic) are used 100% of the time.

The basic optimization techniques applied to the code sequences are

1. reorder instruction sequences to remove pipeline interlocks,
2. pack together instruction pieces into a single MIPS instruction
3. remove the effects of delayed branches

In some cases it may be necessary to insert no-ops to prevent illegal pipeline interactions or to accomodate delayed branches. Also, pieces of instructions may be left blank whenever no piece is available to pack with the instruction.

The reorganization problem is discussed in detail in another paper[5]; the problem is shown to be NP-complete and a set of heuristic solutions is proposed. The reorganization algorithm is essentially an instruction scheduling algorithm. The basic algorithm is

1. Read in the program in assembly language and create a dag indicating precedence scheduling relationships among the instructions.
2. Determine which groups of instructions can be scheduled for execution next and eliminate the others.
3. Heuristically choose an instruction to shedule from the executable instructions. Attempt to choose an instruction that can be packed with the last instruction executed and that will allow the rest of the code to be scheduled with a minimum number of no-ops.

The reorganization problem is made difficult but the potential presence of overlapping resource utilization in parallel code streams. This overlap must be detected before scheduling of either stream occurs; once it is detected, a deadlock state where neither stream can be scheduled for execution is avoidable. These reorganization techniques (without the instruction packing) can obtain performance improvements of 5-10% over code that must wait for completion of a previously dependent instruction. The use of instruction packing increases the relative effectiveness of this reorganization.

The optimization of delayed branches is the control-flow conterpart of code reorganization. Our algorithm for branch delay optimization examines the targets of the branch in an attempt to obtain useful instructions to execute during the delay time. The branch delay algorithm[6] can obtain space and time improvements in the range of 10-20% for the MIPS branch instructions.

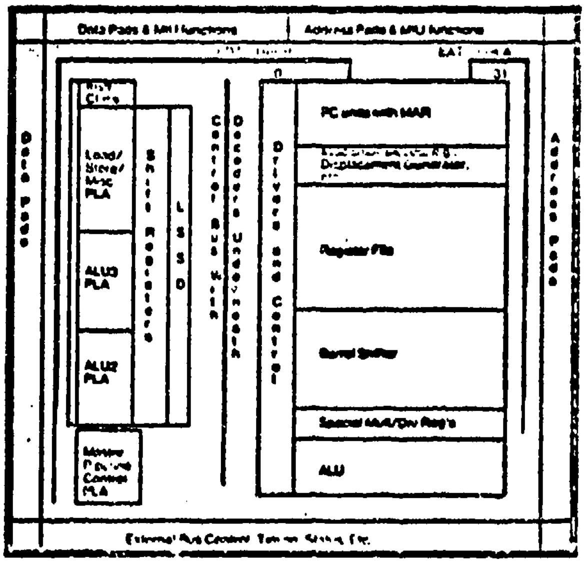
Present status and conclusions

The entire MIPS processor has been laid out and partitioned into a set of six test chips that cover all the data path and control functions on the chip. Four test chips have been sent out for fabrication as of August 1982; we expect send the remainder to fabrication during August 1982.

In the software area, code generators have been written for both C and Pascal. These code generators produce simple instructions, relying on a pipeline reorganizer. A complete version of the pipeline reorganizer is running. An instruction level simulator is being used to obtain performance estimates.

Figure 2 shows the floorplan of the chip. The dimensions of the chip are approximately 6.9 by 7.2 mm with a minimum feature size of 4 μ (i.e. $\lambda = 2\,\mu$). The chip area is heavily dedicated to the data path as opposed to control structure, but not as radically as in RISC implementation. Early estimates of performance seem to indicate that we should achieve approximately 2 MIPS (using the Puzzle program[7] as a benchmark) compared to other architectures executing compiler generated code. We expect to have more accurate and complete benchmarks available in the near future.

Figure 2: MIPS Floorplan



The following chart compares the MIPS processor to the Motorola 68000 running the Puzzle benchmark written in C with no optimization or register allocation. The Portable C Compiler (with different target machine descriptions) generated code for both processors. The MIPS numbers are a close approximation of our expected performance.

	Motorola 68000	MIPS
Transistor Count	65,000	25,000[1]
Clock speed	8 MHz	8 MHz
Data path width	16 bits	32 bits[2]
Static Instruction Count	1300	647
Static Instruction Bytes	5360	2588
Execution Time (sec)	26.5	6.5

Acknowledgments

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680. Thomas Gross is supported by an IBM Graduate Fellowship.

Many other people have contributed to the success of the MIPS project; these include Judson Leonard, Alex Strong, K. Gopinath, and John Burnett.

An earlier version of this report appears in the Proceedings of the CMU Conference on VLSI Systems and Computations, 1981.

References

1. Radin, G., "The 801 Minicomputer," *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems,* , ACM, Palo Alto, March 1982, pp. 39 - 47.

2. Patterson, D.A. and Sequin C.H., "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. of the Eighth Annual Symposium on Computer Architecture,* Minneapolis, Minn., May 1981, .

3 Lampson, B.W., McDaniel, G.A. and S.M. Ornstein, "An Instruction Fetch Unit for a High Performance Personal Computer," Tech. report CSL-81-1, Xerox PARC, January 1981.

4. Widdoes, L.C., "The S-1 Project: Developing high performance digital computers," *Proc. Compcon,* IEEE, San Francisco, February 1980, .

5. Hennessy, J.L. and Gross, T.R., "Code Generation and Reorganization in the Presence of Pipeline Constraints," *Proc. Ninth POPL Conference,* ACM, January 1982, .

6. Gross, T.R. and Hennessy, J.L., "Optmizing Delayed Branches," *Proceedings of Micro-15,* IEEE, October 1982, .

7. Baskett, F., "Puzzle: an informal compute bound benchmark", Widely circulated and run.

[1] The 68000 IC-technology is much better, and the 68000 performs across a wide range of environmental situations. We do not expect to achieve this clock speed across the same range of environmental factors.

[2] This advantage is *not* used in the benchmark. i.e. the 68000 version deals with 16 bit objects while MIPS uses 32 bit objects

Testing Chips using ICTEST Version 2.5

John Newkirk, Rob Mathews, and Irene Watson: language design
John Newkirk, Irene Watson, and Doug Boyle: compiler
Rob Mathews, Irene Watson, and Wayne Wolf: tutorial

Information Systems Laboratory
Stanford University
VLSI File #012782

ABSTRACT

ICTEST is a superset of the **C** programming language that specializes in test programs for integrated circuits. It is used by the designer to specify the stimulus to a circuit and the expected response from the circuit. ICTEST provides a common interface to a variety of testing and simulation environments, and frees the designer from translating test algorithms into bit vectors. This document aims to provide you, the designer in the street, with enough information to simulate and test chips. Since ICTEST is based on **C**, the prospective user must have a rudimentary knowledge of that language as well. The first section is provided for those with no experience programming in **C**. Experienced **C** users can start with section II.

January 27, 1982

Testing Chips using ICTEST Version 2.5

John Newkirk, Rob Mathews, and Irene Watson: language design
John Newkirk, Irene Watson, and Doug Boyle: compiler
Rob Mathews, Irene Watson, and Wayne Wolf: tutorial

Information Systems Laboratory
Stanford University
VLSI File #012782

1. Minimal C

C shares many common characteristics with the Algol-like and Pascal-like languages. The user familiar with Pascal should pick up the elements of C quickly. Readers familiar only with FORTRAN are to be pitied for their lack of worldliness, but their main difficulty should be in absorbing the more robust control structures offered by C. However, a few *caveats* are in order for all neophytes:

- Case is always significant — Foo is not equivalent to foo. Reserved words in the language are expected to be in lower case.
- There are no procedures in C, only functions. Any procedure may return a value. However, it need not return a value, and the value returned by a function can be ignored.
- There is no general notion of scope, as there is in Pascal. The only scopes are global and within a function. Functions cannot be defined within functions.

1. A simple program

C programs are simply collections of functions. The main program body is defined by a function called *main*. The function definition consists of two parts. First, there is the function name, followed by the list of parameters in parentheses (the parentheses must be present even if there are no arguments), and a declaration of the types of the parameters. Then comes the function block, which declares the local variables and the actions to be taken by the function. The functions may be declared in any order in the program file, but it is wise to keep them in some logical order.

Consider this program:

```
/* This is a comment; comments do not nest */
main ()
  {
        int i, answer;

        for (i = 1; i != 11; i = i+1)        /* compute n! for n=1,...,10 */
          { /* Here begins a compound statement */
                answer = factorial (i);
                wastetime ();
          }
  }
```

```
int                 /* factorial returns an integer value */
factorial (arg1)
  int arg1;         /* its argument is an integer */
  {
          int i, n; /* variables local to the function */

          /* Compute (arg1)! by iteration; 0! = 1 */
          n = 1;
          /* for (initial value; continuation condition; increment) */
          for (i = 1; i <= arg1; i = i+1)
            {
                    n = n*i;
            }
          return (n); /* Return a the result */
  }

wastetime () /* has no parameters, returns nothing */
  {
          int i, counter;
          /* waste time by counting from 1 to 1000 */
          for (i = 1; i <= 1000; i = i+1);
  }
```

The astute reader will note that this program has been rendered uninteresting by the lack of input or output. This situation will be remedied soon; meanwhile, consider several other characteristics of the program. First, integer variables, 32 bits long, are declared by the *int* statement. This is the type most useful for ICTEST programs.

Second, the value to be returned by a function is indicated by the statement

```
return (value);
```

where *value* can be any expression. Also, since *factorial* returns a value, it must have a function type. If the function type is not specified, it is assumed to be *int*. (The function type specification precedes the function name.) *Wastetime*, on the other hand, does not need to return a value, so it does not use the *return* statement, and its call looks much like a procedure call in other languages.

Third, compound statements are made using { and }. Although all simple statements must be terminated by a semicolon, compound statements need no such punctuation.

Fourth, although it is not obvious from this example, all function arguments in **C** are value parameters. Their value for the caller cannot be changed by the callee. C has a mechanism for allowing reference parameters that can be changed by the callee, but it is error-prone and rather baroque. For the purposes of ICTEST, returning a single value with *return* should be sufficient.

Finally, there is the *for* loop. This is one of several control statements available in C. Control structures are the next topic of discussion.

2. Logical expressions and control structures

There are three control structures of interest: the for-loop, the while-loop, and the if-then-else statement. Each evaluates an expression, and depending on its value, may execute a statement (which can also be a compound statement).

Expressions can be used to compare variables and constants, much as you would expect. The comparison and logical operators are:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
&&	logical and
||	logical or
!	logical not

Note that the equality operator is ==. This is often confused with =, the assignment operator. However, **C** allows assignment operations within comparison expressions. Therefore, if you use = when you meant ==, the **C** compiler will probably not complain. This mistake is a common source of infinite loops for beginning **C** programmers.

The *for* loop was demonstrated in the sample program. The general form is

```
for (initialization; while-condition; increment)
  action;
```

Before the loop starts, the initialization statement is executed. Then the condition is tested, and if it is true the action is performed. After the action the increment statement is executed, and the program loops back to retest the condition. Unlike FORTRAN, and like Pascal, a *for* loop can execute zero times if the loop variable is initialized to a value that does not satisfy the condition.

The *for* loop is best understood as a special case of the *while* loop. **C** expands the *for* loop into this *while* statement:

```
initialization;
while (condition)
  {
        action;
        increment;
  }
```

This has the obvious interpretation: while the condition is true, the statement (or compound statement) following is executed.

The final control statement is the *if-then* statement. It can be written with or without an *else* clause:

```
if (condition)
  statement;

if (condition)
  statement;
 else
  statement;
```

This statement again has the obvious interpretation. However, the *if-then-else* syntax differs from Pascal: if the *if* action is a single statement, there must be a semicolon between it and the *else* clause. This is because the semicolon in **C** is a statement terminator, not a statement separator as in Pascal.

3. Arrays

C also allows arrays of one or more dimensions. The declaration for a one-dimensional array of eight integers and a 4 by 5, two-dimensional array of integers is

```
int a[8], b[4][5];
```

The lower bound of an array is fixed at zero, so the elements of a are numbered 0,1,...,7. There is no bounds checking on array indices, so be sure the index is always in the proper range. Also, due to the parameter passing mechanism of C, arrays are an exception to the call-by-value rule. Array parameters can be modified by the callee (actually because C passes a *pointer* to the first element of the array). A function cannot normally return an array unless it is passed in to it. As a rule, stick to one-dimensional arrays where possible; there are some tricky intricacies involving higher-dimensional ones.

4. Macro preprocessing

C programs are passed through a macro preprocessor before they are compiled. The preprocessor performs a number of useful functions: for instance, since C does not have constants in the language definition, the preprocessor can be used to substitute constant values for their names. The three most used features of the preprocessor are *define*, *include*, and *ifdef /endif*.

Define allows the definition of a pseudonym for a string. For example,

```
#define GOO 23
```

will replace all instances of GOO with the string 23 in all lines after the definition. The # marks the line as a preprocessor command, and must be in the first column. Case is significant in the preprocessor, as it is in C; it is traditional for program constants to be entirely capitalized. Constant definition is the prevalent use of *define*, but almost any string replacement can be made.

```
#define begin {
#define end }
```

could be used to make the Pascal programmer more at home, but don't use it – why exacerbate your typing chores.

Include performs file substitution. The command:

```
#include "standard.h"
```

will cause the preprocessor to insert the text of the file standard.h before the next line in the source. Included files may contain *include* statements, but the recursion cannot continue indefinitely.

Include is particularly handy for information that is common between several program files. Only one copy of the data is necessary, and when it is updated all programs that point to it by *include* will receive the update. The extension ".h" is conventional for such header files. They should contain only definitions, data structures, and type declarations, not C code – separate compilation and loading exists for the latter purpose, unlike in Pascal.

Ifdef allows conditional compilation. If-then and if-then-else conditionals are possible. The three commands used are:

```
#ifdef identifier
#else
#endif
```

Ifdef checks whether the identifier has been defined in a previous *define* macro. If so, the source lines between *else* (if present) and *endif* are ignored. If the identifier has not been defined source lines between the *ifdef* and *else* or *endif*, whichever comes first, are ignored. The usual method of using *ifdef* is to define the different conditions with names and head the source file with a *define* macro that sets the desired condition name to have a nonzero value.

5. Input and output

Finally, we come to I/O. I/O procedures in **C** are implemented as standard procedures and macro definitions. A program using the I/O package must include the preprocessor command

```
#include <stdio.h>
```

before any I/O calls are made. (Pointy brackets enclosing a file name indicates the file is in a system library known to the preprocessor.) This call can always be included without cost.

The simplest I/O functions to use are *printf* for output and *scanf* for input. The form for both functions is:

```
printf (command-string, argument, argument, ...);
scanf (command-string, argument, argument, ...);
```

The arguments are the values to be printed or input. Arguments to printf are value parameters, and therefore may be arbitrary expressions. The command string is much like a format statement in FORTRAN: it is a literal string with embedded commands showing the position and type of variables to be printed. Integers are the main item of interest for **ICTEST**, which are described by *%d*. An example command string for *printf* is:

```
"The values range from %d to %d\n"
```

When the output is performed, the first %d is replaced by the value of the first argument in the *printf* call and the second %d is replaced by the value of the second argument. The \n indicates a magic character, as in the editor; in this case it is a newline. *Printf* does not output newlines unless requested to by the \n. The other useful magic character is \t, a tab. The *printf* call using this command string might be

```
printf ("The values range from %d to %d\n", variable1*6, i);
```

Printf performs no checking to insure that the argument type agrees with the type specified in the command string, nor does it check to see if the number of arguments equals the number of values required by the command string. Beware.

Scanf uses the command string to indicate the format of the input requested. Input of two free-format integers can be accomplished by

```
scanf ("%d %d", &variable1, &i);
```

The expression "&variable1" means "address of variable1"; passing such a *pointer* to variable1 allows *scanf* to set the value of variable1, circumventing C's usual call-by-value semantics.

6. Bit hacking

Since **ICTEST** programs exercise chips that understand only bit streams, it is sometimes useful to manipulate integers as bit vectors. There are several **C** operators defined on integers:

&	bitwise and
|	bitwise or
^	bitwise exclusive or
<<	left shift
>>	right shift
~	one's complement

Avoid the temptation to use these operators in a manner dependent on the machine word length. Also, the effect on the sign bit of right shift is undefined in the language; on the VAX, it extends the sign bit.

7. Lint

Lint is a command that examines **C** source programs to detect a number of bugs and obscurities. It enforces the type rules of **C** more strictly than the **C** compiler. *Lint* accepts multiple input files and library specifications, and checks them for consistency. Suppose there are two **C** source files, *file1.c* and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages about various error-prone or wasteful constructions that, strictly speaking, are not bugs.

II. ICTEST

The purpose of **ICTEST** is to provide an algorithmic environment for testing integrated circuits and to free the designer from dealing directly with bit vectors. The designer specifies the stimulus to a circuit and its expected response. He then compiles the **ICTEST** test description and uses it to exercise *either* a simulation of his design or an actual part.

An **ICTEST** program has 2 major parts: declarations of *IOports*, and the test procedures themselves. An IOport describes a logical connection to the circuit to be tested, including the physical pins or simulator nodes and the timing and data formatting involved. The test procedures manipulate the IOport driving and sensing values, and **ICTEST** converts these commands into the sequence of 0's and 1's necessary for the actual test.

ICTEST is tailored for testing two-phase, synchronous designs. It generates the clocking automatically, and it recognizes the two-phase timing types, *e.g.*, **valid phi1**. This tutorial will concentrate on such testing, although **ICTEST** can cope with unclocked or asynchronous designs as well.

Keywords in **ICTEST** are shown in boldface. "Pin" is used generically to refer either to a pin on an IC or a node in a simulation.

1. Declaring IOports

ICTEST exercises the circuit through IOports. For testing physical chips, these refer to pins or groups of pins; for simulating, they can refer to any electrical node. The IOport declaration defines the port name, the data format it expects, and the timing required to drive or read it. The general form of this statement is:

$$\left\{\begin{matrix}\textbf{input}\\ \textbf{output}\end{matrix}\right\} \mathit{IOname} \left\{\begin{matrix}\textbf{activelo}\\ \mathit{nil}\end{matrix}\right\} \left\{\begin{matrix}\textbf{serial}\left\{\begin{matrix}\textbf{lsb}\\ \textbf{msb}\end{matrix}\right\}\mathit{length}\\ \textbf{segs}\left\{\begin{matrix}\textbf{lsb}\\ \textbf{msb}\end{matrix}\right\}\mathit{length}\ \textbf{by}\ \mathit{width}\\ \textbf{parallel}\ \mathit{width}\end{matrix}\right\} \left\{\left\{\begin{matrix}\textbf{stable}\\ \textbf{valid}\\ \textbf{qual}\end{matrix}\right\}\left\{\begin{matrix}\textbf{phi 1}\\ \textbf{phi 2}\end{matrix}\right\}\right\};$$

IOname is the port name, *width* is the number of pins it has, and *length* indicates the number of consecutive clock cycles required to transfer a value to or from the port. **Serial** implies a width of one; **parallel** implies a length of one. Finally, the clocking phrase declares the timing type of the port.

Here are two sample IOport declarations:

```
input Datain activelo parallel 5 stable phi1;
output Dataout serial lsb 24 stable phi2;
```

Conventionally, the first letter of a IOport name is *captalized* to make it easy to distinguish from a C variable. *Datain* is a 5-bit input port, accepting values from -16 to +15, whose values are stable φ_1. It uses negative logic. *Dataout* is a serial port producing a 1-bit-wide, 24-bit-long bit stream, least significant bit first, with values stable φ_2. When *Dataout* is read, the ICTEST will automatically count out the 24 clock cycles necessary to read the port and convert the result into an integer.

If a group of pins operates as a three-state port, the group must be defined with two IOport declarations: one to declare its input characteristics and one to declare its output characteristics. Both ports are then bonded to the same pins (see below).

A qualified clock IOport (**qual** timing type) produces a *single* clock pulse when requested to output a 1. Normal input ports latch their values; qualified clock ports do not.

A segmented IOport is simply a serial port more than 1 bit wide. For example,

```
input Segsin segs msb 4 by 5 valid phi1;
```

declares a port 5 pins wide and 4 time periods long, for a total of 20 bits of data. **Msb** indicates that the most significant segment is presented first. Data is valid on φ_1.

2. Declaring a clock

Normally, you will declare a clock IOport:

$$\textbf{clock}\ \mathit{IOname} \left\{\begin{matrix}\textbf{activelo}\\ \mathit{nil}\end{matrix}\right\};$$

e.g.,

```
clock Clock;
```

Clock (*not* the same as **clock**) is the port name. If *Clock* has 2 pins associated with it (see below), ICTEST generates a 2-phase, non-overlapping clock signal to drive it. A 1-pin clock port is assumed to define φ_1 only.

3. IOports and physical reality

The **bond** statement associates pins with IOports. It must precede the first IOport declaration. It is the one part of the program inherently tied to the target system: for physical chip testing it relates pin numbers and IOports, while for simulators it relates node numbers to IOports.

A **bond** statement for a tester target contains the number of pins on the chip package, the number of pins in the tester socket, and a list of IOport names and their associated pins. The bonding declaration for the IOport in the previous section might be:

```
bond pkg 40 socket 40
  {
          Datain = 3-5,12-11;
          Dataout = 2;
          Clock = 8,9;
          Vdd = 1;
          Gnd = 3;
  }
```

The **pkg** and **socket** information must be present for tester targets, and absent for simulator targets. *Datain* consists of the 5 pins 3, 4, 5, 12, and 11, where 3 is the most significant (sign) and 11 the least significant bit; *Dataout* consists of the single pin 2. Since *Clock* has 2 pins, ICTEST will generate a 2-phase clock.

Vdd and *Gnd* are the pre-defined power and ground input ports. They must be bonded in the last two clauses in the **bond** statement. Either of these ports may contain any number of pins. No IOport declarations are required for these ports.

Pins may be bonded to more than 1 IOport, necessary for 3-state pins, and handy when one port is a subset of another. For example:

```
tri_in, tri_out = 10-18;
parity = 36;
full_data = 11-17, parity;
```

The **bond** statement for a simulator target contains a list of IOport names and their associated nodes. Nodes may be specified as node numbers or as node names. If node names are specified on the right side of a bonding clause, these names must appear in your simulator symbol file, *i.e.*, the *.sym* file. Size information may *not* follow the word **bond**.

As for tester targets, the Vdd and Gnd ports must be bonded in the last two clauses of the **bond** statement. The simulator will then drive the power and ground nodes using information contained in the *.sym* and circuit description (*.sim*) files.

Since it is desirable to use the same program for both simulation and testing of a circuit, and since the **bond** statement must be different for these two types of targets, clever programmers will put their bond statements in separate files from the rest of the test program (*e.g.*, in "bond.sim" and "bond.chip"). The symbol SIM is automatically defined for simulator targets; CHIP, for tester targets, so these files may be conditionally included in the ICTEST program:

```
#ifdef SIM
#include "bond.sim"
#else
#include "bond.chip"
#endif
```

4. Test steps

A test step is an **ICTEST** operation on a IOport. There are 3 basic types of test steps: driving an IOport to a value (input to the circuit), sensing and comparing the value of a port to an expected value (output to the circuit), and sensing and storing the value from an output port (also output to the circuit). As will be discussed later, there is also an enforced "NOP", requiring no activity on a port for a specified number of clock cycles (the **pad** test step).

All test steps have the format:

```
IOport(s) operator value(s);
```

Value(s) may be absolute numbers, variable names, function calls, or C expressions. Expressions must be enclosed within parentheses. Test steps are normally executed one after another in the order that they appear in the program on consecutive clock cycles.

4.1. Drive test step

Driving a port to a value is similar to variable assignment:

```
IOport(s) = value(s);
```

The IOports are driven to the value(s). If more than one value is present, the values (from left to right) are driven to the port in successive clock cycles. If more than one IOport is present, each is driven to the values. Both lists are comma-separated.

A port can be driven to any value that will fit into its word length. Data is converted from 2's complement integers into bit vectors and formatted for the port. Recall that if the port is declared as **activelo**, the bits are complemented before being sent to the port. For example,

```
Datain = 7;
```

drives the port *Datain* to the value 7, and:

```
Datain = 7, 8, 9;
```

drives it to 7, then 8, then 9 on successive clock cycles.

4.2. Sense test step

Sensing the value of a port is equally simple. The format for this test step is:

```
IOport(s) =? value(s);
```

The output(s) of the IOport(s) are compared to the data value(s) on the right side, analogously to the corresponding *drive* step. For example,

```
Dataout =? convolve(i);
```

will cause *Dataout* to be read and compared to the value returned by *convolve(i)*. If the values do not match, ICTEST produces a diagnostic indicating the value expected and the value actually read. If the compare is successful, no message is output.

4.3. Store test step

The test step:

```
IOport => storeloc;
```

senses and stores the value at an output port. Only a single port name and a single store location (variable name) may be specified.

Storing occurs at the end of a test or subtest, *not* immediately! Be sure to read the sections on a sample test program and on literals (coming up) to understand when you may usefully use the stored value. *Storeloc* may be a scalar variable or a one-dimensional array element. The variable or array must have been previously declared with a **result** declaration:

```
result storeloc;
```

For example, a scalar and a 10-entry array could be declared as:

```
result bus_value;
result bus_values[10];
```

and used in a store test step:

```
Bus_A => bus_values[3];
```

Result locations are automatically defined to be 32-bit two's complement integers. Values sensed at lower-precision ports are sign-extended to 32 bits.

The test steps above look very much like C assignment and expression evaluation statements. They are not. IOports are not variables and cannot be passed to functions as arguments or used inside expressions.

5. A sample test program

These statements are sufficient to write a test program for a simple chip. Imagine a chip that takes as input a 16-bit data value, and outputs the square of the data on the next clock cycle (it uses a good algorithm). An appropriate **bond** statement is:

```
bond pkg 40 socket 40 /* only good for a tester */
  {
        Datain = 6-10,12,14,16; /* input x */
        Squareout = 25,28-34;   /* output x**2 */
        Clock = 4,5;
        Vddin, Vddout = 10;
        Gndin, Gndout = 21,1;
        Vdd = 10;
        Gnd = 21,1;
  }
```

If you wish to sense if power and ground are properly connected (a good idea), you must declare ports (*e.g.*, *Vddin* and *Vddout* here) and bond them to the same pins as Vdd and Gnd. Vdd and Gnd are special ports and may not be sensed or driven.

The IOport declarations are next, followed by the test.program:

```
input Vddin parallel 1;
output Vddout parallel 1;
input Gndin parallel 1;
output Gndout parallel 1;
input Datain segs lsb 2 by 8 stable phi1;      /* The data is read and written */
output Squareout segs lsb 2 by 8 stable phi2; /* as 2 bytes, lsbyte first */
clock Clock;

/*
 * test() is always the main program; <stdio.h> is #included automatically
 */

test()
  {
      int i;

      /* power and ground connections are checked here */
      Vddout = 0; Vdd = ? 1; Vddout = 1;
      Gndout = 1; Gndout = ? 0; Gndout = 0;

      /* first test a subset of the possible input */
      for (i = 1; i != 65; i++)
        {
            Datain = i; /* input the data to the port */
            Squareout =? square(i); /* function square defined below */
        }
      /* now let the user specify an assortment of numbers, ending with 0 */
      while ((i = getnum()) != 0)
        {
            Datain = i;
            Squareout =? square(i);
        }
  }

int
getnum()
  {
          int i, nscanned;

          printf ("number to test? ");
          nscanned = scanf ("%d", &i); /* read the number to square */
          return (nscanned == 1 ? i : 0);
  }

int
square(n)
  int n;
  {
          return (n*n);
  }
```

In this program, *getnum* is called to get values to place into the test vector. When the user types a zero (or anything else that *scanf* rejects), *getnum* returns 0, the *while* loop exits, and the *test* completes. The test is then run, and any miscompares from the sense test steps are displayed.

Note that **ICTEST** automatically formats the data for the segmented input and output ports and, since a clock is declared, makes the clock tick at the right times to communicate with the chip. Recall that the *Datain* and *Squareout* IOports are segmented 16-bit ports, formatted as two 8-bit bytes in two consecutive clock cycles.

The following activity will occur during the first 9 clock cycles of the program's first loop. ''–'' indicates no activity for that port during that clock phase; however, remember that input values persist until new values are assigned.

Clock Cycle	φ1	φ2	*Datain*	*Squareout*
0	1	0	–	–
	0	1	1 (lsb seg)	–
1	1	0	–	–
	0	1	1 (msb seg)	–
2	1	0	–	–
	0	1	–	–
3	1	0	–	–
	0	1	–	1 (lsb seg)
4	1	0	–	–
	0	1	2 (lsb seg)	1 (msb seg)
5	1	0	–	–
	0	1	2 (msb seg)	–
6	1	0	–	–
	0	1	–	–
7	1	0	–	–
	0	1	–	4 (lsb seg)
8	1	0	–	–
	0	1	–	4 (msb seg)

6. Compound test steps and concurrency

The statements presented so far have their limitations. For instance, there is as yet no way to present simultaneously data and a ready signal to the circuit.

To control concurrency, you may group test steps. Such a compound test step defines a testing *context*, an environment with such attributes as sequential or concurrent test step execution and generation or non-generation of two-phase clocking. When the *test()* function is entered, clocking is on if a clock is declared, and execution is sequential. Thereafter, whenever a new compound test step is entered, the context attributes are stacked. Completing a compound test step generates a return to the previous context environment.

There are 2 types of compound test steps, concurrent and sequential. In a concurrent compound test step, all test steps are started simultaneously:

[[test steps and other C statements]]

In a sequential compound test step, the test steps are executed sequentially:

{{ test steps and other C statements }}

Compound test steps may be nested, allowing very complex control and sensing. For example, using 2 pairs of sequential brackets inside a pair of concurrent brackets results in 2 test sequences starting simultaneously and executing concurrently.

The default clocking is whatever prevails in the surrounding context. However, a leading clocking phrase can turn clocking on or off:

```
clocked   [[ /* clocked regardless...   */ ]]
unclocked [[ /* unclocked regardless... */ ]]
```

Any ICTEST function, not only *test()*, may contain test steps. Such function is called a *test function*. Unfortunately, when a test function is called, a new context is *not* created. Therefore, if a test function is called from within a concurrent context, its test steps will be concurrent; if from a sequential context, sequential. You probably do *not* intend such a result! The safest approach is to include all test steps in any function within a compound test step; that way the context is explicit.

Use *return* and *goto* statements carefully within testing contexts. In particular, control must flow out the end of the context if the previous context is to be recovered. For example, *never* nest a *return* statement inside a compound test step within a test function.

7. Time frames and offsets

A test vector describes actions on all pins for a series of time frames. In a clocked context, a clock cycle covers 2 time frames, 1 per phase. In an unclocked context, the equivalent of 1 clock cycle is 1 time frame. If a serial, length 4, input port were referenced in an unclocked context, the port would be driven for 4 consecutive time frames. In a clocked context, the same port would see new data every second time frame over 8 time frames (*i.e.*, over 4 clock cycles, with driving during the correct phase).

The execution of a test step in a compound test step can be relocated in time by using an offset label. The offset label is a number or expression in parentheses followed by a double colon, preceding a test step. The test step is delayed by that amount from when it would have occurred were no offset label present. For example, if you want to reset the circuit and then, after 79 clock cycles have passed, check the output of the random number generator, you could write:

```
{{
        Reset = 1;
        Reset = 0;
 79::   Random =? ran_gen(seed);
}}
```

Reset goes high on the first clock cycle of the sequence and low on the second clock cycle; then, after 79 idle cycles (the third through 81st), the result is checked. Equivalently, you might write:

```
[[
        {{ Reset = 1; Reset = 0; }}
 81::   Random =? ran_gen(seed);
]]
```

or perhaps:

```
    [[
            resetchip();
     81::   Random =? ran_gen(seed);
    ]]

  resetchip()
    {
      {{
            Reset = 1;
            Reset = 0;
      }}
    }
```

An expression can also be used as an offset label, as long as it is surrounded by parentheses:

```
(RANSTART-1):: Random =? ran_gen(seed);
```

The compare will be delayed RANSTART-1 cycles from when it would occur were the offset label not present.

Currently, offsets may only occur on test steps and test function calls, not on compound test steps. This restriction is easy to sidestep, however, by using test functions like *resetchip()* above to contain a compound test step that should be offset. The offset can then be placed on the call to such a function.

In an unclocked context, the offset specifies the number of time frames to delay the test action.

8. Iteration and pipelining

Prior to examining how pipelining is accomplished in **ICTEST**, let's look at an example that uses iteration but does not involve pipelining. (You should assume that all IOports in the examples in this section have been declared as parallel and that the enclosing context is clocked.) Suppose a new value and a control flag are input to the circuit every 4 clock cycles. Suppose also that a value is output every 4 clock cycles, delayed 3 cycles from the control flag. You might write:

```
{{
        for (i = 0; i != 64; i = i + 1)
          [[
                {{ Control = 1,0; }}
                Datain = i;
            3:: Dataout =? outfunction(i);
          ]]
}}
```

The loop repeats 3 test steps in a concurrent context. The loop itself is within a sequential context, which means that each iteration of the loop executes in sequence. (If this loop were in a concurrent context, all iterations of the loop would execute simultaneously.) The loop repeats every 4 clock cycles, since the longest test step takes 4 cycles (the output test step includes 3 cycles of delay and 1 cycle for the output).

For each iteration of the loop, the *Control* port is set to 1 then 0 in the first 2 clock cycles. It remains unchanged from 0 for the third and fourth clock cycles. The *Datain* port is set to *i* in the first clock cycle and remains unchanged for the second through fourth clock cycles. The output is ignored on the *Dataout* port until the fourth clock cycle, when it is compared to *outfunction(i)*.

The sequential brackets surrounding the test step driving the *Control* port are necessary, since all actions in a concurrent compound test step are started simultaneously. If the sequential brackets were not present, **ICTEST** would produce an error announcing that the user attempted to assign two values to the same port at the same time.

To force the loop to repeat in more than 4 clock cycles, you must use the **pad** test step. **Pad** steps are NOP's, forcing no activity to occur on an IOport for some number of clock cycles. For example:

```
Control = pad 5 ticks;
```

will add 5 cycles of padding to the last requested operation on the port *Control*. (In an unclocked context 5 time frames of padding would be added.) Similarly,

```
Control = pad 2;
```

will pad the control by twice its declared length. Thus, if *Control* were a serial port 12 bits long, no operation would be allowed within 24 clock cycles following the latest activity on *Control*. The amount to pad may be any C expression.

This modified loop iterates every six clock cycles:

```
{{
        for (i = 0; i != 64; i = i + 1)
          [[
                {{ Control = 1,0; Control = pad 4 ticks; }}
                Datain   = i;
            3::  Dataout =? outfunction(i);
          ]]
}}
```

Now no activity occurs on any of the ports in the fifth and sixth clock cycles of each iteration.

Now suppose that the design is pipelined, so that new input data can be accepted before the previous output is complete. To test such a part completely requires the use of a **pack**-ed context. The **pack** phrase can precede any compound test step and, when present, specifies that all test steps within the context are to be started as soon as possible without overlapping any test steps (including **pad** test steps) from the previous iteration. Thus, for:

```
pack {{
        for (i = 0; i != 64; i = i + 1)
          [[
                {{ Control = 1,0; }}
                Datain   = i;
            3::  Dataout =? outfunction(i);
          ]]
     }}
```

each iteration of the loop is 4 clock cycles long and begins 2 clock cycles prior to completion of the previous iteration. In other words, the loop begins another iteration every 2 clock cycles, and any 2 consecutive iterations of the loop are overlapped in time by 2 clock cycles. Note that no test steps are actually overlapped: no activity is specified for *Control* in the third and forth clock cycles of each iteration; no activity is specified for *Datain* in the second to fourth clock cycles; and no activity is specified for *Dataout* in the first three clock cycles.

We can summarize the resulting activity for the first 8 clock cycles in a table. "–" indicates no activity for that port during that clock cycle. *Datain* and *Control* are assumed to be valid on φ_2.

Clock Cycle	*Control*	*Datain*	*Dataout*
0	1	0	–
1	0	–	–
2	1	1	–
3	0	–	*outfunction(0)*
4	1	2	–
5	0	–	*outfunction(1)*
6	1	3	–
7	0	–	*outfunction(2)*

If the part accepts new data every 3 clock cycles rather than every 2, the control stream may be padded to reduce the amount of pipelining (with **pad** or just with a 0):

```
pack {{
         for (i = 0; i != 64; i = i+1)
           [[
                  {{
                     Control = 1,0;
                     Control = pad 1 ticks;
                  }}
                  Datain   = i;
              3:: Dataout =? outfunction(i);
           ]]
     }}
```

A context is always unpacked unless a **pack** phrase appears.

9. Literals

Literals are commands meant to be passed directly to the simulator or tester target. The only messages currently of interest are for one of the simulators (*esim* or *tsim*):

{S *message*; S}

A common example is a simulator initialization command such as:

{S iu; S}

Literals have the side effect of dividing a test into pieces: to synchronize the literal command with the testing commands, ICTEST must run the test specified so far, then issue the literal command. The current test vector is sent to the testing target, the test is run, the results are returned, and the error messages (if any) are printed. Finally, the literal is processed, and a new test vector is begun. This procedure is used regardless of whether the literal is intended for the current target.

This side effect is useful because ICTEST can only handle a limited-length test vector. The maximum length depends upon the particular target, and a longer test will generate the error message "out of tester memory" (or possibly "allocation error" or "reallocation error"). If you get one of these messages, use a literal simulator command (with null contents) to chop up the test.

Literals may not be nested inside a compound test step or inside a function that is called from within a compound test step. Also, you should not expect dynamic storage to be preserved while processing a literal — flushing the test vector takes a long time.

10. Printing messages

Printf and all other C I/O procedures are executed during the *creation* of the current test vector rather than during the *execution* of the test itself. Hence, all *printf* output occurring during a subtest will precede any **ICTEST** error messages. Consequently, **ICTEST** provides special print routines, *eprintf* and *mprintf*, that are synchronized with errors from the test itself.

When a compare error occurs, **ICTEST** first prints the most recent *eprintf* message prior to the sense test step, unless this message has already been printed. The **ICTEST** error message is then output. *Mprintf* messages are always printed (m = mandatory). *Eprintf* and *mprintf* are otherwise identical to *printf*.

For example,

```
for (i = 0; i != 64; i = i + 1)
  [[
        eprintf ("input was %d, expected output was %d\n", i, i*i);
        X = i;
        Y = i;
        Out =? i * i;
  ]]
```

will print an *eprintf* message followed by an error message each time the actual output does not match the expected output. If you use a *printf* instead of an *eprintf* statement, however, you will get 64 *printf* messages followed by any error messages detected by the test. If you use an *mprintf* instead of an *eprintf* statement, you will get 64 *mprintf* messages with any error messages interleaved between them at the point where errors occurred.

11. ICTEST diagnostics

The error messages output by **ICTEST** when comparison errors are detected have the form:

*E *bin-value oct-value dec-value*	*expression*
W *bin-value oct-value dec-value*	*filename, linenumber, portname*

"E" is the expected value and "W" the actual value, displayed in binary, octal, and decimal. *Expression* is the right side of the sense test step that caused the error (suppressed if the expression was merely an absolute number). The *filename*, *linenumber*, and *portname* are self-explanatory.

12. Simulation

To prepare for simulation, you follow exactly the same steps as you would were you intending to use *esim/tsim* manually. See *esim(1)*, *extract(1)*, and "A Guide to Design Validation for EE271" by Wayne Wolf for more information. Note that if you specify node names in your **ICTEST bond** statement, these names, as well as the names *vdd* and *gnd*, must be defined in your *.sym* file.

Your **ICTEST** program should normally begin by initializing the simulator:

{S iu; S}

Deal with "*x* nodes unknown" problems by providing the appropriate argument to *iu*.

Another useful command,

```
{S *; S}
```

drops you into the simulator's command interpreter. The simulator will prompt you as usual, and you may issue any simulator commands that seem appropriate, ending with a request to exit the interpreter and continue the test.

13. Testing

There is no particular magic to using the **MINIMAL** and **MEDIUM** testers. See "The **MINIMAL** and **MEDIUM** Testers" by Rob Mathews for more information.

Appendix A — ICTEST test vector display

ICTEST provides a language for describing the stimulus to and expected response from a device under test. Logical values are presented to and sensed on pins; these logical values can be organized as parallel or serial streams of bits and are typically associated with a specific phase of the clock.

The data array that contains the test pattern is called a *test vector*. Its width is the number of pins for the device, and its length is usually twice the number of clock cycles in the test (2 phases per clock cycle). *Literals* are not reflected in the test vector, since they delimit subtests.

If a compiled **ICTEST** program is run with the −d option, the test vector is directed to the terminal instead of the target tester or simulator. This appendix explains the test vector display.

The display begins with the number of pins under test and the physical pin numbers, node numbers, or node names as column headings for the test vector. The vector is organized by IOports, ordered left to right in the order they appear in the bonding map. The next line gives the type of each pin:

I = input, active high	i = input, active low
O = output, active high	o = output, active low
T = three-state, active high	t = three-state, active low
C = φ_1 clock, active high	c = φ_1 clock, active low
D = φ_2 clock, active high	d = φ_2 clock, active low
Q = qual φ_1 clock, active high	q = qual φ_1 clock, active low
R = qual φ_2 clock, active high	r = qual φ_2 clock, active low
V = Vdd	
G = Gnd	

The test vector follows. A line of all plus symbols indicates a *flush*, *i.e.*, the end of a subtest. All lines (vector rows) between the flush lines are sent to the target as one vector.

Each line of the display represents a row of the test vector (a time frame). The symbols that appear in the vector are defined as follows:

Symbol	Meaning
d	Drive stable input pin to 0
D	Drive stable input pin to 1
v	Drive valid input pin to 0
V	Drive valid input pin to 1
s	Sense if pin is 0
S	Sense if pin is 1
c	Drive clock pin to 0
C	Drive clock pin to 1
q	Drive qualified clock pin to 0 for this time frame only
Q	Drive qualified clock pin to 1 for this time frame only
w	Wait until pin is 0
W	Wait until pin is 1
−	No activity for pin
P	Pad − required no activity for pin for this time frame
x	Error − conflicting actions for pin for this time frame

0s and 1s are physical, not logical, values at this point, so they will be inverted if the port that contains the pin is **activelo**.

At the end of each line is error handling information. It permits **ICTEST** to reconstruct the original logical value and to issue appropriate error messages. For each pin sensed during this vector line, three numbers (separated by commas) and a code will appear to the right of the test vector line:

1) an index to the relevant prototype error message in the message array following the test vector,
2) a unique identifier for this particular value,
3) the bit position in the sensed value of this sense request (the least significant bit position is numbered zero), and
4) a code that indicates whether this is the final bit of this value and whether the data is active-high or active-low:

Code	Meaning
f	Final bit, data active-high
space	Not final bit, data active-high
F	Final bit, data active-low
underbar	Not final bit, data active-low

For example, a single test line might be:

```
-dDcCss-- 0,5,14  0,5,15f
```

meaning: the first, eighth, and ninth pins in vector are idle; drive the second pin low; drive the third pin high; clock the fourth pin low (φ_1, since first clock pin is always φ_1); clock the fifth pin high (φ_2); and sense the sixth and seventh pins and expect low values.

At the end of the line, since two pins were sensed, we find two error handling displays. Both displays point to error message 0, and since both contain the unique identifier 5, they are two bits of the same value expected from a port. The sixth pin in the line is sensed for bit 14 of the value and the seventh pin in the line is sensed for bit 15 (the final bit) of the value. The value is active-high since the code in the first display is a space, and the code in the second display is an *f*. (It can also be inferred from these two error handling displays that the sensed pins belong to an IOport declared as segmented 8 by 2 with lsb output first and valid on φ_2 – bits 0-13 were probably sensed in the preceding lines.)

The first test vector is the initialization vector, automatically generated by **ICTEST**. The final test vector is terminated by a line of asterisks.

The line after the asterisks specifies the number of prototype error messages. Each prototype error message includes the filename and line of the relevant test step in the source code, and the original expression from which the test value was computed. An example of a prototype error message is as follows:

```
test1.ict  29  0  1  i*i
```

i*i is the expression from which the expected value for the output port was computed, on line 29 of *test1.ict*. The output port is indicated by the fourth entry in the line, the 1. This number is an index into the IOport names that were declared in the **bond** statement, counting from 0; it is the second bonded port in this example. The third entry in the line is 0 for a sense test step and 1 for a store test step.

An example of a prototype error message pertaining to a store test step is:

```
test1.ict 29 1  1  i
```

The value sensed at the output port will be stored into the variable *i*. The other information is as above.

A CLOCKING DISCIPLINE FOR TWO-PHASE DIGITAL SYSTEMS

David Noice, Rob Mathews, and John Newkirk

Dept. of Electrical Engineering
Stanford University

ABSTRACT

A two-phase clocking discipline for digital ICs can guarantee freedom from clock skew, critical races, and other timing problems. In this paper, we present such a discipline: a notation, composition rules, and early results from verification tools.

INTRODUCTION

Sooner or later a designer of digital circuits must face the problems of clock skew, critical races, and hazards. Generally he does a timing analysis that uses the tedious and error-prone process of calculating delays and deriving timing diagrams to check for errors. Alternatively, he may perform timing simulations to test a circuit for a representative set of inputs, a process that may leave errors undetected.

This paper describes a different approach: to guarantee correct operation despite uncertainty about delays in the circuit. The result is a clocking discipline that deals with timing abstractions only. It is not based on delay calculations; it is only concerned with correct, synchronous operation at some clock rate*.

The examples we present are drawn from nMOS IC design. However, most of the ideas are applicable to any two-phase digital system.

THE CLOCKING DISCIPLINE

This clocking discipline is a method for producing circuits that are free from timing errors. It consists of two main parts: a notation for signal types, and composition rules for legal circuits. In effect, the notation and rules define a syntax of clocking-correct circuits that can be checked by auditing tools, analagous to the syntax and type checking done for computer programs.

A design that follows the discipline is guaranteed to have no errors due to clock skew, critical races, or hazards. A more accurate simulation results: a circuit that is free from timing errors can be simulated without fear of faulty race predictions.

Assumptions

The clocking discipline rests on three basic assumptions:

1) All input and initial values are digital.
2) The system is two-phase and synchronous.
3) All logic and wiring delays are positive and bounded (less than some fixed maximum value). No knowledge of relative delays among circuit paths is used.

The first assumption is important. The discipline guarantees continued digital operation, but it needs an initial starting point and digital inputs.

The second requirement has two bases. First, a two-phase clocked system is a very practical method for designing most MOS ICs[1]. Second, and more importantly, it can be shown that two phases are needed to avoid critical races if relative delays between different paths are unknown[3].

The final assumption arises from the goal of immunity to clock skew and races. To achieve this goal the discipline must insure a design will work under all possible circuit delay conditions. Accordingly, we must not assume any given circuit path is faster or slower that any other path. This assumption is especially important for IC designs where wiring delays can potentially be a dominant factor and are unknown until layout is finished.

This assumption gives rise to another constraint as well. To insure finite delays and digital signals the class of allowable combinational logic must be restricted. Combinational logic cannot have any unclocked feedback loops. Such loops can potentially oscillate, with no bounded settling time.

Notation

A notation of signal types and clocks derives from these assumptions. To motivate the notation, consider the dynamic latch shown in Figure 1. The input signal must satisfy timing requirements with respect to the clock (φ_1) to ensure that the pass transistor will properly latch the input value. In particular, the input must be valid in a window around the falling edge of φ_1, providing setup and hold times required by the latch. Such a signal becomes valid on phase 1, denoted $v\varphi_1$.

Figure 2 illustrates a representative set of clocking types. Clock types (φ and qualified φ) are signals that establish the time and sequence

This research was sponsored by DoD ARPA under contract MDA 903-79-C-0680.

*This philosophy is also shared by self-timed asynchronous circuits [1,2].

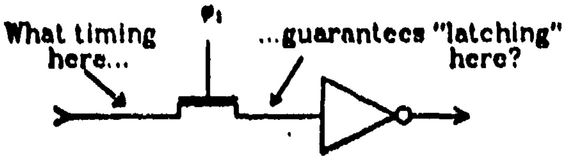


Figure 1. A φ_1 clocked dynamic latch.

references for the data types (valid φ and stable φ). The two sets (clock and data) are separate. A clock signal is never a valid data signal. This reflects the fact that a clock cannot satisfy the setup and hold time requirements with respect to another clock without a race.

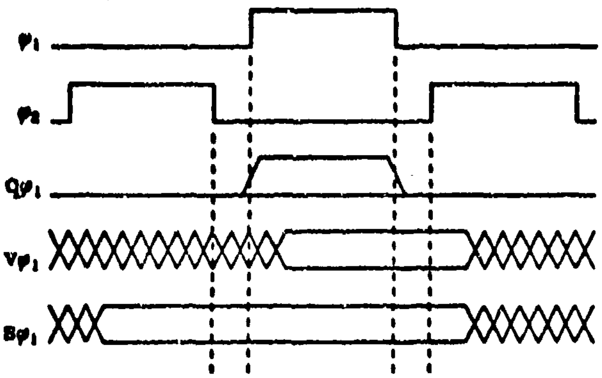


Figure 2. Some representative clocking types, showing the sequence of events. Q, v, and s stand for qualified, valid, and stable.

Note that all the signal types are invariant given arbitrary but bounded skew and delay. For example, a $v\varphi_1$ signal is valid some delay after φ_1 rises. By stretching the φ_1 period, we can guarantee that the $v\varphi_1$ signal is settled before the setup time required by a φ_1 latch. Similarly, by stretching the gap between φ_1 falling and φ_2 rising, we can guarantee sufficient hold time in the face of clock skew.

Composition Rules

Now let us investigate some of the properties of these signals. With a little thought we can see that none of the data signals change types when they go through combinational logic. Furthermore, combinational logic that has all its inputs $s\varphi_1$ will have outputs that are $s\varphi_1$ (see Figure 3a).

A memory element formed by a clocked pass transistor and inverter *will* change the signal type (see Figure 3b). If the input is $v\varphi_1$ or $s\varphi_1$, the output is $s\varphi_2$. This is the foundation of the two-phase clocking discipline. φ-clocked storage nodes are time-bounding elements. They are the points where values must have settled in time, and where values are held (and in some sense converted) for the opposite phase.

Note that an $s\varphi_1$ signal may not change value during the entire φ_1 period. When an $s\varphi_1$ signal is ANDed with a φ_1 or $q\varphi_1$ clock (see Figure 3c), we get an output that fits the definition of a $q\varphi_1$ clock. The output cannot glitch. It either mimics a φ_1 clock or remains low as shown in Figure 2.

The properties described above form composition rules for building clocking-correct circuits. They are the basic rules we can use to create a syntax of legal circuits.

A Simple Clocking Discipline

A simple clocking discipline can be constructed with only the six timing types φ_1, φ_2, $q\varphi_1$, $q\varphi_2$, $s\varphi_1$, and $s\varphi_2$ and the composition rules shown in Figure 3. A legal circuit comprises an interconnection of memory elements and combinational logic such that all inputs to memory elements receive signals of the proper types: clocks to the clock inputs and stable data signals to the data inputs. These structuring rules are equivalent to requiring that the circuit can be two-colored[4], one color for φ_1 and $s\varphi_1$ paths, another for φ_2 and $s\varphi_2$. All together, the notation and rules define a syntax of clocking-correct circuits (see Figure 4).

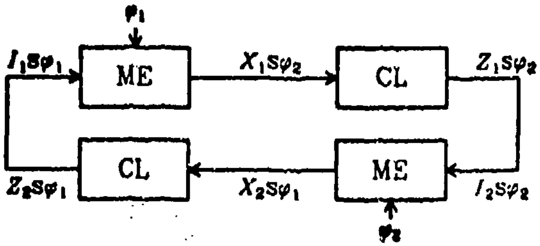


Figure 4. A typical two-phase finite state machine show the syntax and consistency of the clocking types and composition rules.

An analysis of the small example in Figure 5 will give the reader some feel for the utility of the

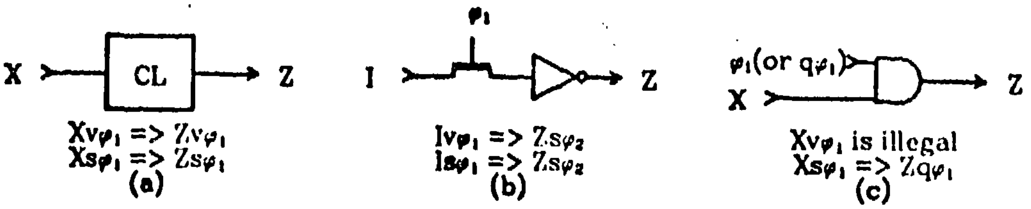

X
CL
Z
$Xv\varphi_1 => Zv\varphi_1$
$Xs\varphi_1 => Zs\varphi_1$
(a)
φ_1
I
Z
$Iv\varphi_1 => Zs\varphi_2$
$Is\varphi_1 => Zs\varphi_2$
(b)
φ_1(or $q\varphi_1$)
X
Z
$Xv\varphi_1$ is illegal
$Xs\varphi_1 => Zq\varphi_1$
(c)

Figure 3. Composition rules for a) combinational logic with bounded delays, b) memory elements, and c) qualified clocks.

clocking discipline. The figure shows a memory element with storage at node 3. The node is periodically driven through T1 or T2. Will this circuit work?

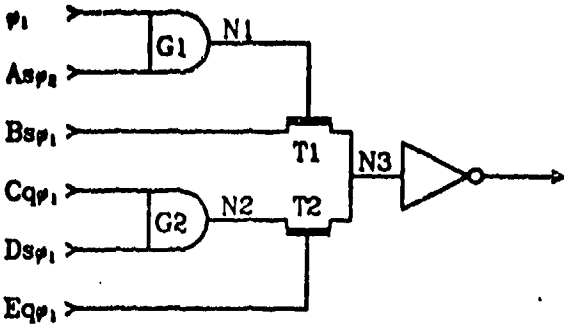


Figure 5. An example for analysis using the clocking discipline.

Let us examine the top branch first. G1 has two inputs: φ_1 and $As\varphi_2$. It does not fit any of the composition rules given in Figure 3, so it is illegal. An $s\varphi_2$ signal can change value several times during the φ_1 period. The output at N1 is a "glitchy" qualified clock that can destroy the value stored at N3.

In the bottom branch, the inputs to G2 do fit the rule shown in Figure 3c, so N2 is $q\varphi_1$. However, a $q\varphi_1$ signal is not allowed as input to T2: only $v\varphi_1$ and $s\varphi_1$ signals may be latched by a memory element. The race between the falling edges of $Eq\varphi_1$ and $Cq\varphi_1$ through G2 makes the value stored at N2 indeterminate.

It should be clear that if signals A and C were both $s\varphi_1$, the circuit would work and fit within the clocking discipline. This example shows some of the usefulness of the discipline. The clocking discipline becomes increasingly valuable as the circuits analyzed become bigger and more complex.

The simple discipline described so far is too restrictive for MOS IC designs. In practice we have found it necessary to include such basic techniques as precharging, sharing of bus lines across clock phases, and feedback for static memory elements. These techniques have introduced new pitfalls (such as charge-sharing), but they are caught by the auditing tools described next.

Auditing Tools

The clocking discipline can help a designer produce cleaner, clocking-correct circuits, but mistakes will still happen. Auditing tools can eliminate those mistakes. These tools look for such errors as clocks used as data, data signals used on the wrong phase, *etc.* They also flag charge sharing, feedback loops, and the like for extra attention from the designer.

Once a design has passed the auditing tools it should be completely free from clock skew and race problems. A unit delay simulator[8] will now adequately predict the outcome of any races because they are not critical races—they have no effect on the final output.

It would be nice if the circuit could also be *exactly* modeled by a switch simulation. Unfortunately, there are still a few remaining problems, *e.g.*, drive fight (a multiplexor that allows two gate outputs to fight each other), and charge leakage off of a storage node.

The clocking discipline and auditing tools could preclude these problems, but the added restrictions would be costly. For example, we could eliminate the danger of charge leakage by requiring all memory elements to be refreshed every clock cycle. Currently, the designer is responsible for avoiding these problems. A switch simulator that could detect these conditions, combined with the clocking discipline, could guarantee a completely accurate functional simulation of a design's operation.

Results

The clocking discipline has been tested on eight student and research IC projects. During the design phase these projects made use of preliminary clocking discipline concepts, but the auditing tools were not yet available. Furthermore, time constraints prevented all but a few from doing even minimal simulation before fabrication.

During the Winter quarter of 1982 the fabricated ICs were received and complete electrical tests and simulations were done. At this point the paper designs were also checked with the auditing tools. The results are shown in Table 1.

Result	number
Clocking, simulation and chip ok	3
Non-fatal clocking error; simulation and chip ok	2
Fatal clocking error; simulation could detect	2
Fatal clocking error; simulation could not detect	1

Table 1. Results of testing eight IC designs with auditing tools, simulation, and electrical tests.

The experience with these designs has demonstrated that the clocking discipline is an effective and practical technique. The auditing tools are particularly useful because many errors can be caught at an early stage, before simulation.

It is interesting to note the types of errors found. In two cases non-fatal timing errors were flagged that pointed out some student misconceptions about precharged logic. Fortunately the mistakes only affected the performance of the chips rather than their operation. Three designs had fatal timing errors. In two of these cases additional simulation could have caught the error.

However, the third case had previously gone through extensive simulation and was expected to work. This design had a critical race. The simulator predicted it would work, but the actual chip failed. After the fact, the auditing tools pinpointed the location and nature of the race. In the of the race. In the future, the auditing tools will be used before simulation and fabrication.

The auditing check only takes about 5 to 10 CPU seconds on a VAX 11-780 for a design with 1000 transistors. The run time increases linearly with the number of transistors.

CONCLUSION

The different facets of the clocking discipline have all proven to be useful. First, the signal notation is a valuable aid in teaching, analyzing and designing two-phase circuits. Second, the composition rules and auditing tools guarantee that a design will be free from timing errors due to clock skew, races, or hazards as well as flagging other potential problems for extra attention.

The clocking discipline is a practical technique that guarantees clocking-correct designs, and more accurate simulations. The result will be reduced design time, fewer design iterations, and cheaper working designs.

REFERENCES

1. Mead, C.A. and Conway, L., *Introduction to VLSI Systems*, Reading: Addison Wesley, 1980.
2. Miller, R., *Switching Theory*, vol. 2, New York: Wiley, 1965, chapter 10.
3. Noice, D., Ph.D. thesis, in preparation.
4. Baldwin, B., "A representation for timing and topology", *Internal memo, LSI Systems Area*, Xerox PARC, January 1979.
5. Baker and Terman, "Tools for Verifying Integrated Circuit Designs", *Lambda*, Palo Alto, Fourth Quarter 1980.

Testing the High Yield Memory

Tim Saxe

ISL, Stanford University
VLSI File #821019

ABSTRACT

The High Yield Memory chip (HYM) was designed at Stanford to test our newest design tools. The chip was explicitly designed to be highly testable and to correct for single bit hard errors. The results of extensive testing of the chips are recorded in this report. The report consists of sections that describe the tests and then sections that describe the results of applying the tests to chips from different fabrication efforts.

October 26, 1982

Testing the High Yield Memory

Tim Saxe

ISL, Stanford University

VLSI File #821019

The initial tests were performed on 12 chips from Mosis run M18N. Of these chips, 4 had storages times of several seconds, 6 had storage times of 500μs, and two had storage times of 50μs. Given the speed of our tester, only the four with storage times over a second were reasonably testable. Of the four "good" chips, one was perfect. Another had a single row that malfunctioned and could be corrected by the error correction circuitry. The other two chips had failures that were serious enough to make the chips useless. On one, the precharge failed and on the other the addressing shift register has some failure that makes it impossible to access all of the memory.

While it was encouraging to have one perfect chip, the results as a whole are disturbing. Firstly, the great range of storage times (5 orders of magnitude) is not reflected in any of the parameters returned by Mosis. Secondly, many of the testing circuits failed in ways that suggest breaks in relatively short lines (although we haven't been able to see such breaks) and this suggests a high defect density. Finally, the chip was designed to be very testable but it still required a great effort to locate the problems – and we still have no real indication of how good our design was.

Update for run M1DV: this time we had 14 chips that worked out of 30 tested. Of course, a few new mysteries were added to our collection, and have not been resolved.

run	number	dead	no storage time	no defect	correctable
M18N	12	1	8	1	3
M1DV	30	3	5	13	14

Storage time

Since our tester is relatively slow (circa 4 microseconds per pin toggle), storage time is critical to the successful operation of tests. The storage time test consists of loading 1's into a bank of latches in the encoder/decoder, and then gating the output of the latches onto the pads. By using a scope, one can then watch the output pad go high and then, one storage time later, go low.

chip	time
1	7
3	5
4	3
5	8
b1	40e-6
b2	400e-6
b3	400e-6
b4	400e-6
b5	400e-6
b6	40e-6
b7	400e-6
b8	400e-6

Address unit

The address section of the HYM chip is a shift-register. In normal operation a single one is clocked through the shift-register. The column that contains the one-bit is "selected", and can be put in either read or write mode. There are three tests for the address unit.

Test 1 enables all of the pass transistors in the shift-register (E1 = 1, E2 = 1) and thereby converts the shift-register into a chain of 100 consecutive inverters. The basic operation of each inverter, and the continuity of the chain, can be checked by setting the input to the chain to 0 and checking that a 0 is output, and by setting the input to 1 and checking that a 1 is output.

Test 2 clocks the shift-register and sets the input to alternate 1's and 0's. By checking for alternating 0's and 1's at the outputs, the pass transistors can be checked, as well as various timing bugs.

Test 3 sets the shift-register to 0's, sets the input to 1, and then clocks 50 0's through the shift-register. The output is then checked to see if it has 49 0's followed by a 1 and then another 0. This test verifies that the shift-register has the correct length and will perform as required for the operation of the memory.

chip	test1	test2	test3
1	y	y	y
3	n	n	n
4	y	y	y
5	y	y	y
b1	y	y	y
b2	y	y	y
b3	y	y	y
b4	y	y	y
b5	y	y	y
b6	y	y	y
b7	y	y	y
b8	y	y	y

Encoder/Decoder

The encoder/decoder unit has two latches: an in-going latch and an out-going latch. There are two multiplexors: one connects the tri-state pads to one of the error signal, true out-going data or complemented out-going data, and the other connects the word slection multiplexor to either the true in-going or complemented in-going data. There are three tests for the encoder/decoder.

Test 1 checks the basic operation of the latches and their associated multiplexors. This is done by loading a value (all 0's or all 1's) into the in-going latches. On the next cycle, the data is read out by multiplexing it from the outputs of the in-going latches, through the out-going latches and the out-going latches' multiplexor to the pads. This tests that the two multiplexors work to some extent (does not test the operation of the error signal output), that the in-going latches can store data, that all four inverters associated with the latches work and that there are no breaks in the multiplexor control lines (since the multiplexor control lines have output pads on their non-driven ends).

Test 2 checks the operation of the error detection circuitry. This is done by loading the out-going latches with zeros, except for one latch that has a one. The in-going latches are then loaded with zeros, and the multiplexor is switched to output the error signals to the pads. This checks that each exclusive-or gate functions and that the error signal can be multiplexed onto the output pads. In addition, the "ok" line is checked each time to make sure that it is correctly or-ing the outputs of the exclusive-or gates. A side effect of this test is to check, once more that the in-going latches have storage time and also that the out-going latches have storage time. Also, an additional series of tests are performed in which the out-going latches are loaded with all 1's except for a single 0. Finally, a test with no errors is included to verify that the "ok" line is not stuck at 0.

Test 3 checks that the "preio" signal can in fact precharge the word selection multiplexor.

chip	test1	test2	test3
1	y	y	y
3	y	n%	y
4	y	y	y
5	y*	n#	y
b1	--	--	--
b2	--	--	--
b3	--	--	--
b4	--	--	--
b5	--	--	--
b6	--	--	--
b7	--	--	--
b8	--	--	--

-- Invalid due to other errors

* Detected some malfunction in the *compi* line.

% Can't read error values, suggesting that the *erro* line is broken.

# A failure when the error value is 1111101111. This value does not seem to be stored or remembered correctly, but this may be a read-out problem, since the error value is returned correctly. The value read out is 1111100111; error value is correctly 0000010000.

Precharge

The bio lines can be precharged. There are two tests of the precharge logic.

Test1 is a basic test that first allows the bio lines to discharge, and then precharges them. The circuit is considered OK if the bio lines are first 0 and then 1. One test is made for each position of the selector.

Test2 is a diagnostic test, based on a hypothesis that the memory may have some shorts between the column select lines and the bio lines. This test is basically the same as test1 except that column selects are also set to 1's.

Chip	test1	test2
1	y	y
3	y*	y*
4	n	n
5	y#	y%
b1	--	--
b2	--	--
b3	--	--
b4	--	--
b5	--	--
b6	--	--
b7	--	--
b8	--	--

--	Invalid due to errors in previous tests
*	word 4, bit 2 would not precharge
#	word 1, bit 7 would not precharge
%	word 1, bit 7 is directly controlled by read enable

Testing the memory plane

The raw test accesses the memory plane in raw mode (no error correction used). Each word is written as 0's, and read back. This tests for bits that are stuck-at 1. Then, each word is written as 1's, and read back. This tests for bits that are stuck-at 0. The results are plotted to look like the memory plane. There is one square per memory cell. If the cell is empty, it had no stuck-ats. If the cell contains an upward pointing arrow, the cell had a stuck-at 0; a downward pointing arrow, a stuck-at 1.

The error correction test, like the raw test, writes words consisting of ones and then zeros into the memory. Unlike the raw plane tests, the error correction circuitry is activated and should correct any single bit errors.

chip	raw	correcting
1	Y	Y
3	(1)	(2)
4	--	--
5	(3)	y
b1	--	--
b2	--	--
b3	--	--
b4	--	--
b5	--	--
b6	--	--
b7	--	--
b8	--	--

-- Tests invalid due to other problems

(1) Have one row stuck-at 0 (address problems, however)

(2) Fully correct, except that don't have full addressing

(3) One row with a cross-point failure.

Appendix A
Testing suggestions

When starting to test a chip the bond statement may well be incorrect. A few simple tests will pinpoint some problems here. First, check that all input pads can be driven both high and low. Second, check that all output pads are stuck at either 1 or 0. Thirdly, if tri-state enables are externally accessible, test tri-state pads in both input and output modes (also, watch power drain).

Each test should start by checking Vdd, Ground and substrate if used. No point in testing an unpowered chip. Declare Vdd to be *activelo* and reverse senses on tests. Otherwise, vdd will be initialized to 0.

When running simulations, do not initialize the simulation. This will ensure that your test sequence drives every node that should be driven. The presence of X's is a sure indication that your test is incomplete.

Always run you test program against the simulation before trying the real circuit.

On a real circuit, undriven nodes seem to leak to 0, but don't rely on it.

Run your tests from the outside in. Bugs in the outer level have a strange way of fouling up more interior levels.

Write down your test results as you go and keep copies of your test programs. It's surprising how often you will want to go back and retest some outer level — possibly in greater detail.

The tester has a nasty habit of remembering pin values from one test to the next. This may give the appearance of abnormally long storage times. If you run a test that relies on charge leaking off (eg. precharge testing), you should explicitly deactivate the contols and wait a few seconds.

Appendix B
Run M18N, M1DV(c1-c30)

chip	t	a1	a2	a3	c1	c2	c3	b1	b2	raw	ecc
1	7	y	y	y	y	y	y	y	y	y	y
3	5	n	n	n	y	y*	y	y*	y*	y*	y
4	3	y	y	y	y	y	y	n	n	--	--
5	8	y	y	y	y*	y*	y	y*	y*	y*	y
b1	40μ	y	y	y	--	--	--	--	--	--	--
b2	400μ	y	y	y	--	--	--	--	--	--	--
b3	400μ	y	y	y	--	--	--	--	--	--	--
b4	400μ	y	y	y	--	--	--	--	--	--	--
b5	400μ	y	y	y	--	--	--	--	--	--	--
b6	40μ	y	y	y	--	--	--	--	--	--	--
b7	400μ	y	y	y	--	--	--	--	--	--	--
b8	400μ	y	y	y	--	--	--	--	--	--	--
c1	<0.1	--	--	--	--	--	--	--	--	--	--
c2	2	y	y	y	y	y	y	y	y*	y	n
c3	<0.1	--	--	--	--	--	--	--	--	--	--
c4	7	y	y	y	n	n	n	n	n	n	n
c5	4	y	y	y	y	y	y	y	y*	y	y
c6	0	--	--	--	--	--	--	--	--	--	--
c7	4	y	y	y	y	y	y	y	y*	y	y
c8	<0.1	--	--	--	--	--	--	--	--	--	--
c9	7	n	n	n	y	y	y	y	y*	y	y
c10	2	n	n	n	y	y	y	y	y*	y	y
c11	8	y	y	y	n	n	n	n	n	n	n
c12	8	y	y	y	y	y	y	y	y*	n	y
c13	8	y	y	y	y	y	y	y	y*	n	y
c14	7	y	y	y	y	y	y	y	y*	n	y
c15	2.8	y	y	y	y	y	y	y	y*	y	y
c16	8	y	y	y	y	y	y	y	y*	y	y
c17	<0.1	--	--	--	--	--	--	--	--	--	--
c18	4.5	y	y	y	y	y	y	y	y*	y	y
c19	4.5	y	y	y	y	y	y	y	y*	n	n
c20	4	n	n	n	y	y	y	y	y*	n	n
c21	2	y	y	y	y	y	y	y	y*	y	n
c22	6.5	y	y	y	n	n	y	y	y*	y	y
c23	4.5	y	y	y	y	y	y	y	y*	n	n
c24	10	y	y	y	y	y	y	y	y*	y	y
c25	1	y	y	y	y	y	y	n	n	n	n
c26	3	y	y	y	y	y	y	y	y*	y	y
c27	.5	y	y	y	y	n	y	y	y*	n	n
c28	<0.1	--	--	--	--	--	--	--	--	--	--
c29	10	y	y	y	y	y	y	y	y*	n	n
c30	7	n	n	n	y	y	y	y	y*	y	y

A NEW AREA ROUTER, THE LRS ALGORITHM

Lyle R. Smith, Tim Saxe, John Newkirk and Rob Mathews

Department of Electrical Engineering, Stanford University
Stanford, California

Abstract

The LRS is a new algorithm for two-layer routing of rectangular regions (area or switchbox routing). It routes regions with the pins fixed on all four sides and, if it fails to complete a routing, returns an estimate of the area it needs to complete the problem. A new measure for evaluating the difficulty of area routing problems affords a comparison between the LRS and the Lee algorithms; the LRS has uniformly superior performance.

Introduction

In the past, most chip routing has been done using channel routers. (A channel is a rectangular routing region with pins placed on two opposing sides. Nets may run out the other two ends of the routing region, but the pin locations on these sides are not fixed.) In routing custom VLSI we have found that channel routing does not perform acceptably. Custom chips often contain large rectangular routing regions with pins located on all four sides. In addition, these regions often do not have a preferred horizontal or vertical direction. These types of problems are not suited for channel routing algorithms.

Several algorithms have been developed to attack the area routing problem, the Lee[1] and Hightower[2] algorithms being the most well known. In order to analyze the performance of these algorithms, we need to be able to describe area routing problems and to measure their difficulty. In channel routing, problems are often described in terms of their congestion factor and constraint graphs. Algorithm performance can then be measured in terms of how close to the channel congestion factor the algorithm completes the channel wiring. These measures, however, do not directly carry over to the area routing problem.

Properties of an Area Measure

There are several important properties of area routing that an area routing measure must capture. Unlike many channels, where there is no fixed area limitation (nominally, channel width can expand forever), area routing problems typically have a definite fixed size. Thus, instead of talking about a number such as the congestion factor for a problem, one must talk about the problem's intrinsic difficulty. This measure of difficulty should decrease as the area of the problem is expanded. The measure should also be independent of any particular routing algorithm, thereby providing a standard of comparison for all area routing algorithms.

The Manhattan Area Measure

We have developed an area routing measure that has both of the properties mentioned above. Let us assume that we are dealing with a gridded world and that only Manhattan wiring is allowed (the extensions to non-gridded and non-Manhattan wiring are straightforward). The Manhattan area measure is as follows: For each net in the problem, the minimum-length Manhattan path required to route it is computed; this computation is done independently of any other nets. For two-point nets, the computation is the Manhattan distance between the two pins. For nets with more than two pins, it is necessary to compute a Steiner tree for the net. The minimum paths for all of the nets are summed and the result is divided by the number of grid points in the region.

The resultant number represents the fraction of the grid points that *must* be used to complete the routing. Multiple routing layers are accommodated by multiplying the number of grid points in the region by the number of routing layers before dividing it into the path lengths.

A sample computation is shown in Fig. 1.

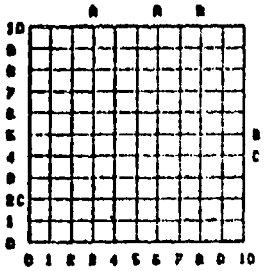


Figure 1. An example of the Manhattan measure computation. The measure = (3 ((from net A) + (2 + 6) (from B) + (10 + 2) (from C))/ (10 * 10 * 2(layers)) = .11 .

For a fixed number of layers, the lower bound given by the Manhattan measure is not always achievable, because it does not account for interactions between nets. An example is shown in Fig. 2.

The Manhattan measure can be used to classify the difficulty of any given area routing problem. As the measure ranges from zero to one, the problem difficulty increases. Note that the upper bound of the measure is three (with only one layer available), however, when the measure exceeds one the problem cannot be completed in the given number of layers.

This research was sponsored by DoD ARPA under contract MDA 903-79-C-0680.

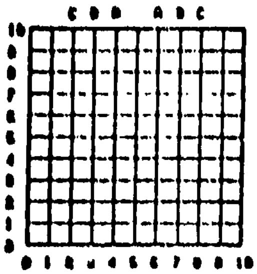


Figure 2. An example of a problem that cannot be solved in the lower bound given by the Manhattan measure with a limited number of layers. The Manhattan measure for two layers is (2 + 4 + 6) / (10 * 10 * 2) = .06. The best two layer solution (assuming wires can run along the edge of the routing region) is (2 + 4 + 8) / (10 * 10 * 2) = .07.

Evaluation of the Lee Algorithm

The Manhattan measure has been used to determine the performance of several area routing algorithms. This is done by generating a series of random examples; the algorithm under test is then run on each example. For a given range of area measures (computed for each test case), the percentage of those that the test algorithm is able to successfully complete is then computed. Fig. 3A shows the completion data for the Lee algorithm.

For those examples where the test algorithm is able to complete the routing, the wire length used by the test algorithm is divided by the product of the grid area and the number of layers available. This *percentage of use* can then be compared to the Manhattan measure to see how efficient the test algorithm was in its use of space. The efficiency of the Lee algorithm is shown in Fig. 3B.

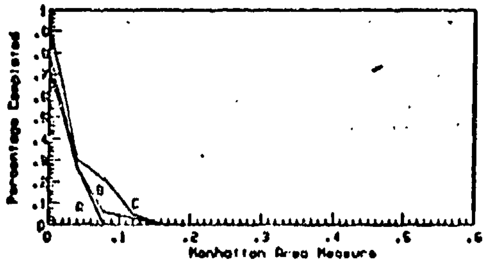


Figure 3A. Performance of the Lee Algorithm. The test problems were 50 by 50 grid points in size and consisted of two point nets. Each curve represents at least 275 examples. In curve A the pins were distributed uniformly along the sides; curve B used a triangular distribution; and curve C used a parabolic distribution.

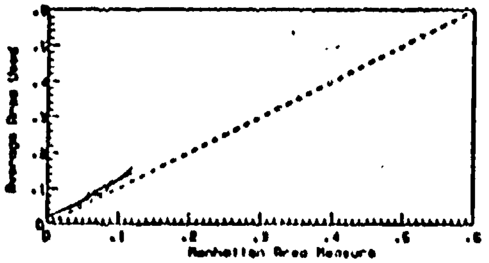


Figure 3B The average area used by the Lee Algorithm when it was successful in routing. The test problems are the same as in Figure 3A.

The Loop Routing Scheme

The Manhattan area measure shows that the Lee algorithm performs quite poorly for general area routing problems. In response we have developed an algorithm, the Loop Routing Scheme, that has much better performance and is also much faster than the Lee algorithm.

The Loop Routing Scheme (LRS) algorithm takes advantage of the "radial" symmetry of area routing problems Wires enter the routing region radially (from the pins) and are connected using circular segments within the routing region. For actual implementation the circular and radial segments are mapped onto a Manhattan geometry system. The basic LRS algorithm will be described assuming a square gridded routing region. Extensions to rectangular regions and non-gridded systems are straightforward

The first step in the algorithm is to assign each net to its own "circular" track. Each track is square in shape and is centered around the middle of the routing region. Succeeding tracks are built growing out from the center of the routing region. The initial layout of the tracks is shown in Fig. 4.

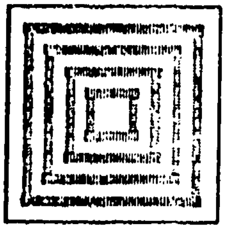


Figure 4. Initial layout of tracks for the LRS algorithm.

A net can be assigned to a particular track only if all of the pins in that net can be connected to the track using a single radial segment. Fig. 5 gives an example.

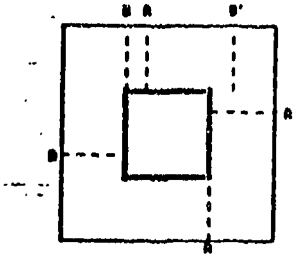


Figure 5. An example of net assignment to a track in the LRS algorithm Net A can be assigned to the track, but net B cannot because the pin B' cannot connect to the track with a single radial segment.

For any given track, there may be several nets that could be assigned to it. The choice of which net to assign from the set of available nets is one of the flexibilities of the algorithm.

If there is not enough room in the initial routing area for all of the circular tracks used, the routing region is expanded from the center. An example is shown in Fig 6. Notice that the expansion is done so that the relative positions of the pins stays the same The new expanded problem can be routed completely in the given area The ability to give an estimate of the area needed to route a problem

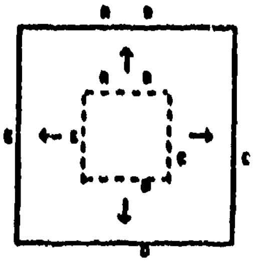


Figure 6. Expansion of the initial routing region by the LRS algorithm. The dotted lines show the original problem and the solid lines show the expanded problem.

is very important when the routing algorithm is to be used in a chip routing system.

After a net has been assigned to a track, all of the pins are connected to the track with radial segments. Segments in the circular tracks connect together all of the radial segments. Here again, there is some flexibility in determining which part of the track is left unused. For example, a simple scheme is to connect pins so as to minimize the path length.

Layer Assignment

After all of the nets have been assigned to tracks and all of the segments have been placed, the routing is as shown in Fig. 7.

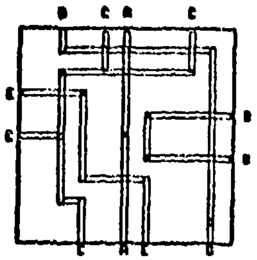


Figure 7. Routing by the LRS algorithm after track and segment assignment, but before layer assingment.

Notice that there has been no layer assignment up to this point. Since all segments are either horizontal or vertical, all horizontal segments can be on one layer and all vertical segments can be on the other. Alternatively, since the LRS algorithm creates tracks in a polar coordinate system, there is another set of orthogonal axes that can be used for the layer assignment. In this coordinate system, all radial segments are assigned one layer and all circular segments are assigned to the other layer.

There are several other colorings that can be considered. For example, one choice would exploit the quadrant symmetry of the problem. The choice of which coloring to use varies both with the problem and the track assignment, and is another one of the flexibilities of the algorithm. Fig. 8 shows an example of a polar coloring.

Comparison to the Akers Algorithm

The organization of the LRS algorithm is similar to that of the unpublished Akers algorithm[3], but there are several fundamental differences. The LRS algorithm is based on a polar coordinate system which guarantees connectability for all pins. The Akers algorithm merely approximates a polar organization and thus sometimes leaves pins unconnected. The LRS algorithm builds the routing from the center of the region. This causes the routing to be packed towards the center of the routing region where there is the most room. This also tends to cause the nets to run on paths as short as possible, instead of running around the outside edges of the region. An additional advantage of routing from the center is that it allows the LRS algorithm to estimate the area needed to route infeasible problems. The Akers algorithm, however, routes from the outside in. This causes it to use longer paths that cannot always be shortened in the cleanup pass.

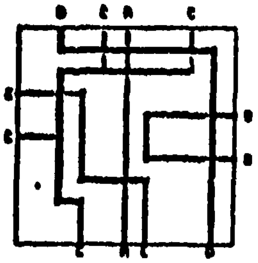


Figure 8. An example of a polar layer assignment for the problem shown in Figure 7.

Other Improvements to the LRS Algorithm

There are several modifications that can be made to the basic LRS algorithm to improve its performance and efficiency. First, multiple pin nets can be broken up into a series of two-point segments that are each routed independently; this introduces doglegs into the nets. Second, multiple segments can be packed into a single track in order to minimize area usage and enhance completion. Third, segments between pins that both lie on the same side of the routing region can be placed in the outer tracks to minimize wire lengths. The algorithm has been implemented with these improvements and a sample routing is shown in Fig. 9.

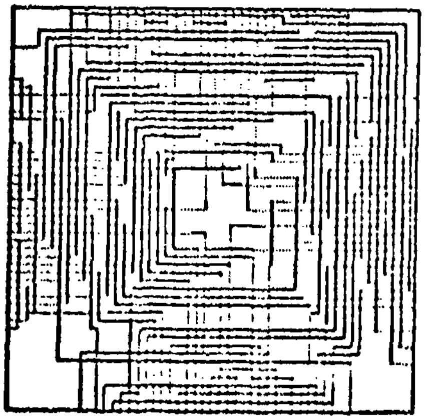


Figure 9. A sample LRS routing which includes doglegs and segment packing.

The performance of the algorithm as measured by the Manhattan area measure is shown in Fig. 10. The figure shows that the performance of the LRS algorithm is quite good compared to that of the Lee.

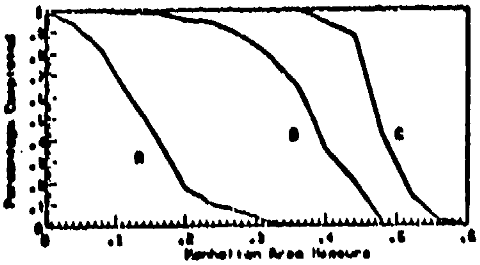


Figure 10A. Performance of the LRS Algorithm. The test problems were 50 by 50 grid points in size and consisted of two point nets. Each curve represents at least 750 examples. In curve A the pins were distributed uniformly along the sides; curve B used a triangular distribution; and curve C used a parabolic distribution.

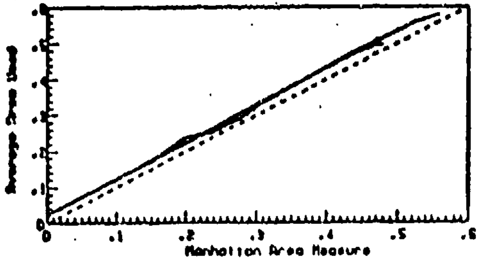


Figure 10B. The average area used by the LRS Algorithm when it was successful in routing. The test problems are the same as in Figure 10A.

The LRS routing scheme can also be extended to doughnut-shaped areas (this type of routing problem occurs frequently when connecting the pads to a chip). The routing tracks are built out from the central obstacle. In this case, wires can connect to the circular tracks from both the outside and the inside of the routing area. Vertical constraints (i.e., opposing pins that can access the same track and interfere with each other) can occur in this scheme and must be accounted for when track assignment is done. The algorithm has been run on pad routing examples with good results (see Fig. 11).

Conclusions

The LRS algorithm has very good performance because it treats the routing problem in polar coordinates. Using polar coordinates yields several important advantages. The algorithm makes better use of its degrees of freedom, running both polysilicon and metal horizontally and vertically. Furthermore, it cannot box itself in, as can the Lee and Hightower algorithms (which are the only truly bidirectional algorithms). Due to the structure of the algorithm, constraint loops never occur. This is a significant advantage over the typical channel routers such as the Dogleg. Finally, although the algorithm is able to perform well for a wide class of routing problems, it is simple to code and fast to execute.

References

(1) Lee, C., "An Algorithm for Path Connections and Its Applications", IRE Trans. E. C. (September, 1961), pp. 346-365.

(2) Hightower, D. W., "A Solution to Line Routing Problems on the Continuous Plane", Proc. Design Auto. Workshop, 1969, pp. 1-24.

(3) Soukup, J., "Circuit Layout", Proc IEEE, Vol. 69, No. 10, Oct. 1981 p. 1297

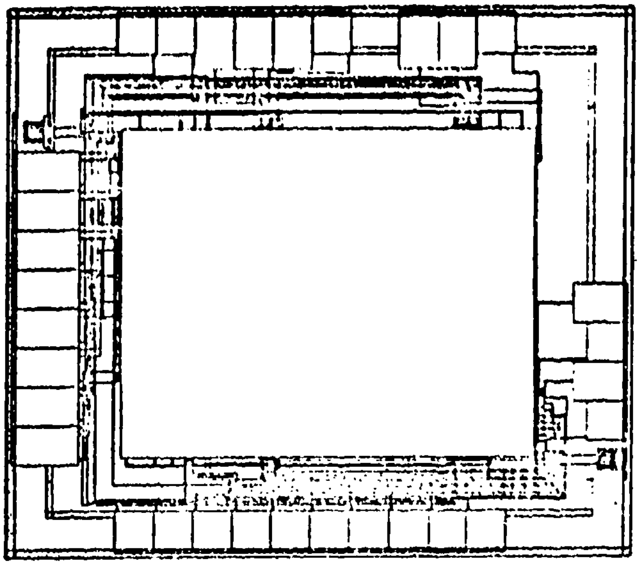


Figure 11. An example of the LRS Algorithm used on a pad routing example.

Heuristic Programming Project
Report No. HPP-82-11
(working paper)

A Framework for Design*

by
Christopher Tong
Stanford University

*This research was done as part of the Knowledge-Based VLSI Design Project, in collaboration with Xerox PARC. The Stanford component of this research is funded by the Defense Advanced Research Projects Agency.

Knowledge-Based VLSI Design Group
Memo KB-VLSI-82-16 (working paper)
Last Modified: April 13, 1982
First Draft: April 13, 1982
Filed on [Indigo]<KBVLSI>Memos>Memo82-16-DesignFramework.bravo

A Framework for Design*

by

Christopher Tong
(Stanford University)

Abstract. Design is the process of creating artifact descriptions that satisfy requirements. Two things can change and evolve during this process: the descriptions and the requirements. This paper decomposes the design process into a search through a goal space and a search through a solution space, conducted by a Problem Designer and a Solution Designer, respectively. Differences between the Problem Designer and Solution Designer are resolved by negotiations. A shallow design theory is used to characterize goal interactions and refinements. Some of these ideas will be tested in *Palladio*, an expert system for aiding VLSI designers.

***This research was done as part of the Knowledge-Based VLSI Design Project, in collaboration with Xerox PARC. The Stanford component of this research is funded by the Defense Advanced Research Projects Agency.**

1. Introduction; design as search.

Design is a dialectic between goals and possibilities (see [5]). Designers consider possibilities in response to their current goals and they refine their goals as they learn what is possible. This dialectic reflects the absence of a complete synthetic theory of design. Designers must begin without knowing exactly what they want or what is possible.

More generally, the dialectic reflects the absence of a complete theory of problem-solving. Design is an exampl: of an *ill-structured* problem (see [3] and [4]); neither the goals nor the possible solutions are well-defined at the outset. AI research has primarily concerned itself with problems whose goals are clear, but whose solutions must be arrived at by first considering many candidates. Studying design forces us to also consider problems whose goals are initially unclear.

Designers often refine or modify their goals as they work. For example, a designer may want to minimize the area of a stack; after considering several possibilities, the designer may perceive that fewer global connections means less area; the designer may establish a new goal, "minimize global connections".

We can augment the common notion that "design is search" by recognizing that both goals and solutions are subject to search. Design can be decomposed into design of solution specifications, by search in a solution space; and design of goal specifications, by search in a goal space. By partitioning the problem space into a solution space and a goal space, we are exploiting the fact that there are two search problems with *separate* knowledge guiding each.

Design alternates between tension and resolution. As designers refine their possibilities and their goals, they often create conflicts: goals may conflict; none of the candidate solutions under consideration may satisfy the current goals. Resolution of such conflicts involves modifying current goals, candidate solutions, or both.

We propose a model of goals in terms of *design dimensions* and *dependencies* among them. We will see how dimensions guide the search in the solution space. We will show how dimensional dependencies guide the creation of new goals and the resolution of goal conflicts.

A model for a design system is proposed in which the different "voices" in the dialectic between goals and possibilities are acted out by different program modules. We have not yet fully determined how these modules interact, but we believe that their interactions are usefully viewed as *negotiations.*

2. Searching the solution space.

By "solution space" we will mean the space of partial as well as complete artifact specifications; solution refinement can occur in such a space. For a given candidate solution some of the current goals may be verifiable, while others may not. Whether goals are verifiable or not has to do with the level of detail of the candidate solution. For example, a goal like "minimize area" is not verifiable for a solution that has not yet assumed physical features.

From the viewpoint of a solution space search, testing candidate solutions against current goals requires two abilities: the ability to refine a solution to a more detailed level, to test it against goals verifiable only at that level of detail; and the ability to modify a candidate that does not meet currently verifiable goals, to try to meet those goals. In standard planning terminology, we need two kinds of *operators* for transforming candidate solutions: operators that *modify* a specification at a given level of detail, and operators that *refine* a partial specification to a more detailed level. Search of the solution space is constrained by the available operators.

When candidate solutions do not meet currently verifiable goals, we have a *solution tradeoff situation*. Solution modification operators help make tradeoffs. For example, having established the goal, "minimize number of modules", we may question whether we have reduced the number of modules as much as possible. We select the operator, FactorCommonModule. This operator takes a module that is shared by several branches of control after a control fork, and "factors it out" before the fork, reducing the total number of modules. Successful application of this operator produces a solution which better satisfies our goal.

How can we select among the available operators? We would like to only consider operators whose effects will probably be compatible with our current goals. With each operator we associate a set of *expected* effects; for example, we can expect FactorCommonModule to decrease the total number of modules, and (unfortunately) increase the complexity of connections. Such effects can be qualified, e.g. the increase in connection complexity can be ignored if the number of factored branches is greater than 3. Expectations provide a rational basis for operator selection; uncertainty of prediction is the price paid.

In summary, search of the solution space alternates between tension and resolution: refinement (to a new level of detail) and modification (to resolve tradeoff situations). Search of the solution space is directed by the expectations of its operators and the current goals.

3. Searching the goal space.

By "goal space" we will mean a space of *goal sets*; this allows us to naturally associate a point in goal space with a solution for which the goals must hold. A given candidate goal set may contain some goals that are verifiable for a particular candidate solution, and other goals that are not. Whether goals are verifiable has to do with the level of detail of the candidate solution. For instance, a goal stated in terms of the area of the stack is well-defined only for stack specifications with physical features, but a goal stated in terms of the number of stack cells can be associated with quite abstract specifications. Knowing that the number of stack cells in a stack strongly influences its area, we can first establish the abstract goal, "minimize number of stack cells" early on (when it is verifiable), predicting this will help satisfy the posted (and currently unverifiable) goal, "minimize area of stack".

We can view goal specifications as *artifacts* to be designed. From this viewpoint, testing candidate solutions against current goals requires two abilities: the ability to refine a goal specification, to introduce new (and possibly verifiable) goals; and the ability to rationally modify a goal specification containing conflicting goals. In other words, we need two kinds of operators for transforming goal specifications: operators that *modify* a goal specification (e.g. relax, tighten, remove, prioritize goals), and operators that *refine* a partial goal specification (e.g. solicit new goals, refine old goals, reduce current goals-posted goals difference). Search of the goal space is constrained by the available operators.

Goals can interact in many ways. When a goal specification contains conflicting goals, we have a *goal tradeoff situation*. The goal set:

{"minimize stack1's operations delay time","minimize stack1's global connections"}

is an example of such a situation; to satisfy the first goal, the stack would have push/pop commands sent to all cells simultaneously, unfortunately conflicting with the second goal. This illustrates an *antagonistic* interaction among goals. Goals can also have *synergistic* interactions. For example, a reduction of area often causes a reduction in delay time.

Goal modification operators help make tradeoffs; for example, the tradeoff situation illustrated above can be resolved by applying the Relax operator to the "minimize operations delay time" goal. ReduceGoalDifference is an example of a goal refinement operator; applying it to the currently well-defined goals, { }, and the posted goal "minimize area of stack1" can result in introducing the currently well-defined goal, "minimize global connections in stack1".

To help us select among operators, we associate with each operator a set of expected effects. For example, we expect that applying the Relax operator to a goal that conflicts with other goals may resolve the conflicts; we also (unfortunately) expect that solution optimality may be degraded if that goal also synergistically interacts with other goals.

To allow us to recognize the different kinds of goal interactions, we propose a model of goals in terms of design dimensions and dependencies among them. A *dimension* is a property of solutions that defines a metric for comparing them (e.g. area). Most design goals can be expressed in terms of cost along a dimension (e.g. minimize area). We view the antagonistic goal interaction illustrated earlier as following from a dependency between the dimensions, OperationsDelayTime and GlobalConnections:

(Antagonistic OperationsDelayTime GlobalConnections).

Similarly, the dimensional dependency:

(Synergistic Area DelayTime)

implies that achieving a reduction in area often results in a reduction in delay time as well.

Dimensional dependencies can also capture knowledge about goal refinement. We have seen how an abstract goal, "minimize number of stack cells" can be established to help satisfy a related, less abstract goal, "minimize area"; this relation follows from the dimensional dependency:

(Abstract NumberOfStackCells Area)

We can enumerate and categorize dimensions and dependencies among them. The context in which they are applicable can be delimited by associating them with certain classes of objects; for example,

(Antagonistic OperationsDelayTime GlobalConnections)
can be associated with the class ObjectWithRepeatingFunctionalComponents. These classes can be arranged hierarchically to allow inheritance of dimensions and dimensional dependencies; for example, the class Stack could inherit from the class ObjectWithRepeatingFunctionalComponents.

Dimensional dependencies constitute a shallow design *theory*, as they help answer the question, "What sets of goals are realizable in artifact descriptions?" Dependency relations like Synergistic and Antagonistic help determine whether currently verifiable goals are concurrently satisfiable. A relation like Abstract helps determine how currently unverifiable goals can be satisfied.

In summary, search of the goal space, like search of the solution space, alternates between tension and resolution: refinement (adding new and more detailed goals) and modification (to resolve tradeoff situations). Search of the goal space is directed by the expectations of its operators and the dimensional dependencies.

4. Coordinating the searches.

We now consider a model for a design system in which the different "voices" in the dialectic between goals and possibilities are acted out by different program modules. These modules include: the Design Theorist, the Problem Designer, and the Solution Designer. The *Design Theorist* stores the dimensional dependencies. The *Problem Designer* searches the goal space. The *Solution Designer* searches the solution space. A solution tradeoff situation arises when the Solution Designer produces a solution that does not meet the requirements set by the Problem Designer. A goal tradeoff situation arises when the Problem Designer produces a set of goals that is not realizable according to the theory of the Design Theorist.

A solution tradeoff situation can be resolved in one of two ways: either the Solution Designer generates another solution that meets the Problem Designer's requirements, or the Problem Designer makes the problem easier. This resolution process can be viewed as a *negotiation* between the Problem Designer and the Solution Designer. The Problem Designer demands evidence that the Solution Designer isn't able to solve the current problem, in exchange for modifications of the current problem (relaxing goals, etc.) Depending on the strength of the evidence, the Problem Designer can: refuse to change the problem; relax the problem a little; or relax the problem so much that the Solution Designer's current "solution" actually solves the new "problem".[1]

Goâls on the *design process* help determine the outcome of negotiations between the Problem Designer and the Solution Designer. Such goals effect the allotment of time and resources to the different program modules. For example, after the Solution Designer has produced a number of unsatisfactory solutions using some of its available resources (e.g. its modification operators), the Problem Designer may make the problem easier, in accord with the goal "minimize design time". We introduce a new program module, the Planner, to store the design process goals. The Planner oversees the negotiations and swings them one way or another depending on goals associated with the design process.

Like the Problem Designer, the Planner can run into conflicts among its goals. Knowing the kinds of interactions that can occur among these goals, we can enumerate them as dimensional dependencies; we can create a "design process theorist" to store them and help guide the Planner, in the same way the Problem Designer is guided by the Design Theorist's dimensional dependencies. The Meta-planner, our final program module, plays the role of "design process theorist", storing dimensional dependencies related to *design process* goal interactions, e.g.:

(Antagonistic "minimize design time" "maximize design quality").

The Meta-planner's dimensional dependencies help the Planner avoid imposing conflicting goals on the design process.

[1] Similarly, responsibility for resolving a goal tradeoff situation can be shared by Design Theorist and the Problem Designer. The Design Theorist will modify its theory in exchange for evidence of counterexamples or new dependencies. This type of negotation is not a central concern of my research.

5. Concluding remarks.

This paper has partitioned design into problem design and solution design. Problem design is directed by the dimensional dependencies of the Design Theorist; solution design is directed by the goals set by the Problem Designer. Design alternates between tension (goal or solution tradeoff situations) and resolution (making tradeoffs). Solution tradeoff situations are perceived as tension between the Problem Designer and the Solution Designer; one or both agents can help resolve the tension, modifying either the goals or the solutions – division of responsibility is determined by the Planner, as it reviews its goals for the design process.

This paper reports work in progress, occuring within the framework of the *Palladio* project (see [2]); the project has thus far described a set of abstraction levels that will make what we have illustrated as a continuous refinement process look more like discrete stages of implementation in lower levels of detail. My research is currently focusing on design at the most abstract levels – thus far, only the representation for a "systems architecture" abstraction level (see [1]) has been implemented.

Thanks to Dan Bobrow and Harold Brown for many helpful comments on earlier drafts of this paper; special thanks to Mark Stefik and Bruce Buchanan for helping to organize the current version.

References.

[1] Bobrow, D. and Stefik, M. *Linked module abstraction: a methodology for designing the architectures of digital systems.* (working paper), Knowledge-Based VLSI Design Group KB-VLSI-81-9, 1981.

[2] Brown, H. *Palladio: An expert system for integrated circuit design.* Submitted to

AAAI Conference, April 1982.

[3] Newell, A. Heuristic programming: ill-structured problems. *Progress in Operations Research*, Vol. 3, J. Aronofsky (ed.), Wiley, New York, 1969, 360-414.

[4] Simon, H. The structure of ill-structured problems. *Artificial Intelligence* 4 (1973), 181-201.

[5] Stefik, M., Bobrow, D., Bell, A., Brown, H., Conway, L., and Tong, C. The partitioning of concerns in digital system design. *Proceedings of the Conference on Advanced Research in VLSI*, Artech House, Cambridge, Mass. 1982.

514

Good Layouts for Pattern Recognizers

HOWARD W. TRICKEY, STUDENT MEMBER, IEEE

Abstract—**A system to lay out custom circuits that recognize regular languages can be a useful VLSI design automation tool. This paper describes the algorithms used in an implementation of a *regular expression compiler.* Layouts that use a network of programmable logic arrays (PLA's) have smaller areas than those of some other methods, but there are the problems of partitioning the circuit and then placing the individual PLA's. Regular expressions have a structure which allows a novel solution to these problems: dynamic programming can be used to find layouts which are in some sense optimal. Various search pruning heuristics have been used to increase the speed of the compiler and the experience with these is reported in the conclusions.**

Index Terms—**Control logic design, dynamic programming, partitioning, programmable logic arrays (PLA's), regular expressions, silicon compilers, string pattern recognition, VLSI layout.**

Manuscript received August 18, 1981; revised January 7, 1982. This work was supported by an NSERC Scholarship, the National Science Foundation under Grant MCS-80-12907, and DARPA Grant MDA 903-80-C-0107.

The author is with the Department of Computer Science, Stanford University, Stanford, CA 94305.

I. Introduction

The design of VLSI circuits is currently a very time consuming operation. Some of the recent work to help alleviate this problem has taken its lead from programming language compiler technology, where great strides have been made by using programs to convert high-level descriptions into lower level programs. The idea of a *silicon compiler* to convert high-level descriptions of circuits into layouts has arisen [1], [4], [5], [11]–[13].

A problem with silicon compilers is the definition of a suitable circuit description language. Some languages are basically descriptions of the upper levels of a hierarchical design. These become "high-level" descriptions when the lower levels of the hierarchy can be derived from libraries and/or a familiarity with the class of circuits being described. The "Bristle Blocks" [5] system is an example of this type of system: it can be used to describe a *data path chip* (registers, shifters, ALU's, etc., built around a data bus).

0018-9340/82/0600-0514$00.75 © 1982 IEEE

A second approach is to use a notation which gives the external behavior required. One method of doing this is to give a sort of program which runs on a machine specified at the register transfer level [11], [13]. This technique is meant to be used for designing computer-like chips. Another notation, which can be used for specifying the controlling logic portion of any chip, is that of *regular expressions*. A regular expression can be used to describe a pattern: a sequence of states in which certain inputs must be seen. One can require that various outputs be given whenever certain patterns have been seen. Some of the many uses of pattern detectors can be found in [8]. This paper discusses a silicon compiler whose input is a regular expression and whose output is a layout for the pattern recognition circuit defined by that expression.

In particular, a way of laying out a circuit for a pattern recognizer in a small area will be described. It is fairly easy to give a programmable logic array (PLA) to implement a pattern recognizer, but a single PLA can be rather large. At the other extreme, one can have logic to recognize each basic symbol of the pattern, joining them up with other logic. Such a method can be proved to yield a layout with an area which is linear in the length of the expression [2], but in practice the resulting layouts have been found to be large. The regular expression compiler uses a network of PLA's, and it gives layouts better than either of the extremes.

The next section will explain how regular expressions represent patterns. Then the implementation of recognizers using networks of PLA's will be described. Numerous networks are possible, so a big part of finding a good layout involves searching for the best (or at least, near-best) division of the expression. The fourth section will discuss how dynamic programming and some judicious heuristics can be used to effect this search. Finally, the last section will give some conclusions, based on experience, about what the various search heuristics can accomplish and how much they cost.

II. Regular Expressions as Patterns

A regular expression is a notation for representing a set of strings of symbols. It is defined recursively as follows.

- The symbol is the most basic kind of regular expression. In the application to circuits, the occurrence of a symbol means that the input wires must be zero or one, according to the symbol definition, within the "current state."
- If E and F are regular expressions, then the *union* $E + F$ is a regular expression which means: either E or F.
- If E and F are regular expressions, then the *concatenation* $E \cdot F$ (or simply EF) is a regular expression which means: E followed by F.
- If E is a regular expression, then the *closure* E^* is a regular expression which means: zero or more occurrences of E.
- If E is a regular expression, then (E) is a regular expression (used for grouping). Unless parentheses are used, the unary operators have precedence over the binary operators, and concatenation has precedence over union.

The following notations, while not necessary, are convenient.

- If E is a regular expression, then the *positive closure* E^{++} is a regular expression which means: one or more occurrences of E. (E^{++} is the same as $E \cdot E^*$.)
- If E is a regular expression, then the *optional occurrence* $E?$ is a regular expression which means: zero or one occurrence of E.

The use of regular expressions to describe pattern recognizers is perhaps best seen by means of an example. The following is the complete input file required by the regular expression compiler for a small example.

```
line data[2]
symbol zero(data[1],-data[2]), one(-data[1],data[2]), any()
;
any (one any* zero + zero any* one)
+ (one any* zero + zero any* one) any
```

The *line* declaration gives the wires that are input to the circuit. A line name can be subscripted (with [· · ·]), as **data** are, to represent more than one wire. One can declare any number of lines. The *symbol* declaration gives the names of the symbols that will occur in the regular expression with the values of the input wires which identify a symbol given in parentheses after its name. Here there are three symbols: **zero,** recognized when **data[1]** is a logical "1" and **data[2]** is a logical "0" (indicated by the "-" in front of **data[2]**); **one,** recognized when the **data** wires are reversed; and **any,** which does not specify either "1" or "0" for the **data** wires, so it is a DON'T CARE. Note that **any** will be recognized at the same time as **zero** or **one**: there is no requirement that the wire combinations for different symbols be disjoint.

The regular expression itself follows the declaration. This one gives all strings of symbols where either: 1) the first symbol differs from the second last symbol, or 2) the second symbol differs from the last symbol. This expression will be referred to as PR2.

The pattern recognizer is a synchronous machine. The successive symbols of a string must appear in successive clock cycles (states) for the pattern to be recognized. Whenever the symbols seen in the preceding states form one of the complete strings specified by an expression, an output signal is given.

The notion of an *expression tree* for a regular expression will be useful later on. The expression tree has symbols as leaves and regular expression operators as internal nodes. It is formed in the same recursive manner that expressions are: the tree for $E + F$ is a node containing "+" with the expression trees for E and F as children; similarly for the other operators. Fig. 1(a) gives the expression tree for $((a + b)^{++})^* \cdot c \cdot (d?)^*$.

A unary operator can be combined with the symbol or operator node beneath it. A cascade of unary operators can be reduced to a single one using obvious rules. This yields a *compressed expression tree*, such as the one shown in Fig. 1(b) for $((a + b)^{++})^* \cdot c \cdot (d?)^*$.

An NFA (nondeterministic finite automaton) can easily be given to implement a regular expression recognizer. In Fig. 2 an NFA to recognize PR2 is shown. Initially, the start state is made *active*. At any time there may be a number of active states. In each successive clock cycle, any active states with transitions marked by a symbol seen in that cycle will make

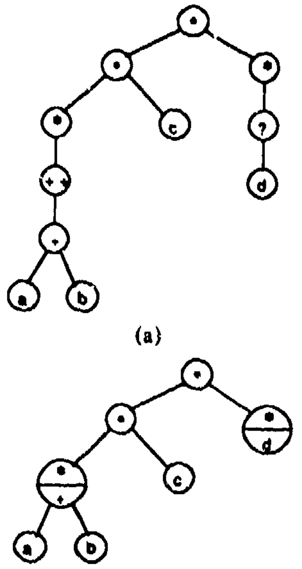


Fig. 1. (a) Expression tree. (b) Compressed expression tree.

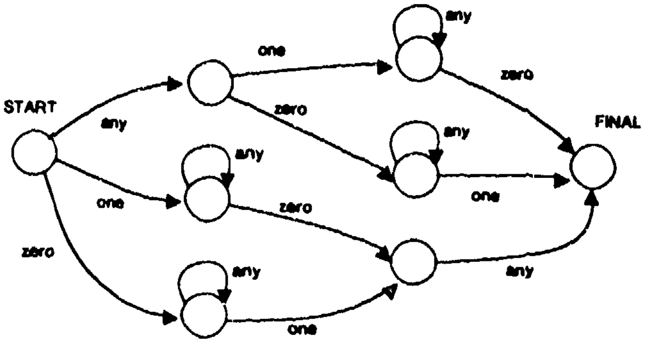


Fig. 2. NFA to recognize PR2.

the successors of those transitions active. States only remain active for one cycle unless explicitly reactivated. Whenever the final state is active, an output signal is given. If desired, the machine can keep operating so that it can detect overlapping occurrences of patterns.

The derivation of an NFA to recognize a pattern is straightforward. For details, see [2].

III. Layout of Regular Expression Recognizers

An easy way to implement a regular expression recognizer is to use a PLA to simulate the NFA corresponding to it. Each state can be represented by a register whose value is calculated by the PLA using the inputs and the current state values (which are fed back from the registers). Details of this method are given in [2].

The problem is that the area used by such a layout will tend to grow quadratically with expression size. A method that leads to a linear growth of the required area is to implement each symbol as a register, together with logic which tests whether or not the symbol is on the input wires. The "symbol modules" have an *enable* input and a *recognized* output. By using appropriate connecting logic, it can be arranged that the symbol modules act like the states of the NFA, where a state is activated by asserting its enable input. (Actually, the circuit is not exactly like the NFA because the state memory is distributed over the transition edges.) It was shown in [2] that as long as the expressions are compressed by combining cascades of unary operators, this method can yield a linear layout. A divide and conquer technique is used to decide where to place the symbol modules and connecting logic. A similar layout would be obtained using the systolic recognizers of [3].

Using individual logic for each symbol gives reasonable layouts, but experience with an implementation of this method has shown that for small expressions the PLA method is better. This is perhaps to be expected, since the regularity of PLA's allows one to pack small numbers of gates more closely than is possible with an *ad hoc* circuit. Thus, the idea of using a combination of the two methods arose. The current implementation of the regular expression compiler uses PLA's for low-level subexpressions, connected together with logic to take care of the operators near the root of the expression tree.

Suppose that one has laid out modules to recognize expressions E and F. It is assumed that these modules are rectangles, and that they have *enable* wires coming in at the left and *recognized* wires leaving at the right. Any input wires required to recognize the symbols in the module's expression must also enter at the left. Then the expressions $E + F$ and $E \cdot F$ can be laid out as shown in Fig. 3(a) and (b), respectively. Operators which have been combined with unary operators can be implemented similarly, as illustrated in the layout for $(E \cdot F)^{++}$ in Fig. 3(c). This type of layout is called an *operator split*. Note that no matter what operator is involved, the two subparts can be laid out either side by side (a *horizontal* split) or one on top of the other (a *vertical* split).

The use of operator splits might be enough to accomplish a layout, but there is the problem that the layouts for the two operand expressions might have very different sizes. This would lead to a lot of white space when a rectangle surrounding the whole layout is defined. The solution to this is to do a *substitution split*. In a substitution split for an expression E, some node D deep in the expression tree for E is replaced by a dummy node. Then the expression rooted at D is laid out (the *dummy tree*), as well as the now smaller expression E (the *father tree*). E will have an *enable dummy* output wire and a *dummy recognized* input wire. The former is attached to the enable input of D and the latter is fed by the recognized wire of D, as shown in Fig. 4.

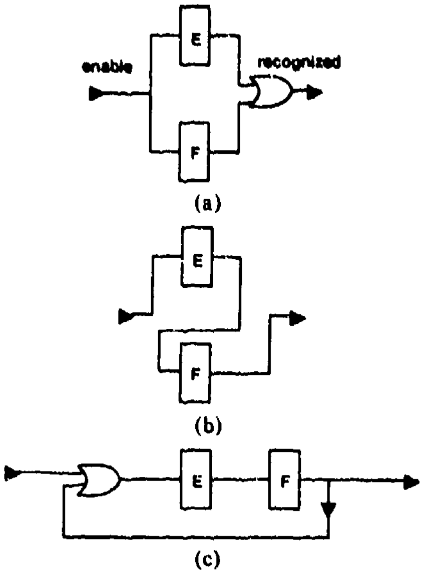
The method for laying out a regular expression, given a compressed expression tree is to either: 1) use a single PLA, or 2) do an operator split or substitution split at the root and recursively lay out the subparts. This accomplishes the goal of using logic to form a network of PLA's for recognizing the regular expression. What remains is to specify how to choose among the various layout strategies. At each stage of the recursion, the following choices must be made.

C1) Should a single PLA, an operator split, or a substitution split be used?

C2) If a split is used, should it be a horizontal or a vertical split?

C3) If a substitution split is used, which descendant expression should become the dummy tree?

One option of the regular expression compiler is to make the above choices guided by the principles that PLA's should be



(a)

(b)

(c)

Fig. 3. Operator splits. (a) $E + F$. (b) $E \cdot F$. (c) $(E \cdot F)^{++}$.

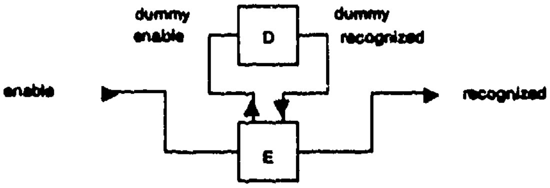


Fig. 4. Substitution split.

neither too small nor too large, and that when splits are used the subparts should be approximately equal in size. In this method splits are performed by looking for a split which yields subparts closest in size, and the recursion continues until the expressions are under some prespecified size. The "size" in terms of area is approximated by the *weight*—the number of leaves in the expression tree.

This heuristic method produces fairly good layouts quite quickly (in approximately 7 s on a VAX/780 for a 150-leaf expression). However, it usually requires some playing around with the parameters of the method to find the best layout possible with this scheme. Even then, a better layout is usually possible. There are several reasons why the heuristic method can be improved upon.

- The idea that two subparts should have the same area is not strictly correct. What really is wanted is for the heights or widths to be about the same. Now, the PLA's generated from regular expressions all tend to have similar aspect ratios (height/width), so that if the subparts are simple PLA's, then the "equal area" principle should hold. It seems plausible that if the subparts are themselves split, then there are some approximately square layouts for them, and so again the equal area principle should yield a reasonable layout. However, an unequal area layout could be even better, and in practice there are many cases where one is better.
- The weight of an expression is only a rough indication of the area needed to lay it out. If the layout involves splits, then the shape of the expression tree affects the economy of the layout.
- The area of a layout depends somewhat on the number of input wires needed. Thus, even if two subparts have equal weights, the layout for one subpart might be taller if it uses more inputs.
- Finally, some optimizations are performed when laying out a PLA (having an effect similar to factoring the expression). This is another reason why the weight of an expression only roughly predicts the area of the resulting layout.

To overcome some of these problems, the regular expression compiler has another option: search systematically through a specified collection of layout strategies, looking for the best one.

IV. Finding Optimal Layouts

An exhaustive search can find the best layout for an expression, given that one is using the general scheme of operator and substitution splits with PLA's at the lowest level. All possible combinations of choices C1), C2), and C3) can be tried, using all possible layouts for the subparts in the case of splits.

Clearly, such an exhaustive search would be very time consuming, even for quite small expressions. One way to avoid a lot of the work is to note that the dimensions of a layout for an expression remain about the same when the layout is made part of a layout for a containing expression. There is often some height increase when a module is incorporated as a subpart in a split because the input wires to the other subpart may have to run through the module. This effect can be calculated, however, so the conclusion is that the strategies for laying out a given subexpression need be calculated only once. The significance of this is that a sort of dynamic programming can be used to effect the search.

Dynamic programming can be used to find optimum strategies for problems that can be broken up as follows. Starting out at a first "stage," some choices are made leading to a collection of smaller, similar problems—the second stage; this continues until some final stage is reached where there are no more choices to be made. If the problem is such that a knowledge of all the optimal solutions at stage i is sufficient to find all the optimal strategies for stage $i - 1$, then dynamic programming can be used. The layout scheme satisfies this condition (approximately), where the problems of stage i are finding the best layouts for subexpressions whose roots are at depth i in the expression tree.

One problem in applying dynamic programming to layout is that one needs more than just the minimum area layouts for the subexpressions: a slightly larger layout may be better to use as a subpart in a split if its height (or width) is closer to that of the other subpart. What is really needed is the best area for all possible heights and widths. In practice this would probably mean keeping all layouts tried, which would eliminate most of the savings that are entailed by the use of dynamic programming.

The solution to this problem is to use an approximation: divide up the continuum of possible aspect ratios into a small number of intervals, and for each subexpression keep only the smallest area layout in each aspect ratio interval.

If the only splits allowed were operator splits, then the search for a layout could follow the standard dynamic programming procedure. Start at the last stage (the lowest leaves) and find layout strategies there; then move up the expression tree, trying single PLA's and operator splits. Trying an operator split is a relatively quick operation, where the dimensions of the children are added to the logic dimensions to give the resulting layout dimensions. (There is also an adjustment for input wires, as mentioned above.)

It is the substitution split which greatly increases the work required to find an optimal layout. After a descendant expression is replaced by a dummy node, optimal layouts have to be found for the father tree. Only some of the layouts found so far can be used: those for subexpressions not involving the dummy tree. Thus, a somewhat independent layout problem must be solved for each possible father tree, and each of those will involve still more father tree layout problems. The work required increases dramatically as the root is approached because there are many more possible father trees (one for each descendant, not including the subproblem father trees).

In fact, by the time all the subproblems have been solved for an expression, layouts will have been found for all possible *prefix trees.* A prefix tree is what is left attached to the root after any combination of descendants have been replaced by dummy nodes.

To get some idea of how many prefix trees there can be, consider T_n, the complete binary tree of n levels. Let S_n be the set of prefix trees of T_n, and N_n be the number of trees in S_n. Any binary tree with $\leq n$ levels is a prefix tree of T_n. A binary tree of $\leq n$ levels can only be formed by having a root with a member of S_{n-1} or the empty tree as left child, and a member of S_{n-1} or the empty tree as right child. Therefore,

$$N_n = (N_{n-1} + 1)^2 \geq 2^{2^{n-1}}.$$

T_n has $m = 2^n - 1$ nodes, so $N_n > 2^{m/2}$. This calculation shows that just enumerating the possible father trees for a balanced expression of 30 leaves (i.e., about 60 nodes) is out of the question.

An obvious partial solution to this is to have some minimum expression size—say 6 leaves—below which an expression will not be considered as a subpart of a split. This has the effect of chopping off some number l of the most populous levels from consideration as dummy tree roots. This changes the above calculation so that now $N_{n-l} > 2^{m/2^{l+1}}$. With this improvement, one could perhaps handle expressions of 30–50 leaves, but it might take a long time, since at the very least one PLA has to be considered for each father tree tried.

To be able to handle expressions with up to, say, 300 leaves, the search needs further pruning. The "equal area" principle mentioned above suggests that splits where one subpart is much bigger than the other are likely to waste space. The regular expression compiler has a *split-ratio* parameter S. Splits will only be considered when the weight ratio of one subpart to the other is in the range $[1/S, S]$. It has been found that in practice $S \approx 2$ yields layouts as good as $S = \infty$.

When all splits are not considered, there turn out to be a large number of subexpressions whose layouts could not possibly be used in the layout for the whole expression. This means that the dynamic programming paradigm of working on the expression tree bottom-up wastes a lot of calculation. It is better to work top-down, looking for subpart layouts whenever required.

To retain the advantages of dynamic programming, a dictionary of layouts is kept so that layouts need never be found twice for the same subexpression. The dictionary can contain layouts for each of the possible prefix trees of each subexpression. This is allowed by having the dictionary indexed by (e, l), where e is an expression node and l is an excision list: nodes that have been replaced by dummies.

Here is the final algorithm for finding layout strategies. There are three tuning parameters to allow trading off search thoroughness for execution time: S, the split-ratio, L, the lowest weight allowed for a PLA, and H, the highest weight allowed for a PLA.

```
FindStrategies($x$:ExpressionTree, $l$:ExcisionList):
  {Find strategies for layout of the expression $x$,
    where the expression nodes on $l$ have been replaced by
    dummies}
      if LookupStrategies($x$,$l$) $\neq$ INIT then return
        {already found strategies for ($x$,$l$)}
      if $x$.weight $\in [L \cdots H]$ then
        TryPLA($x$,l)
      if $x$.lchild.weight/$x$.rchild.weight $\in [1/S \cdots S]$ then
        begin
        FindStrategies($x$.lchild,$l$)
        FindStrategies($x$.rchild,$l$)
        TryOperatorSplit($x$,$l$)
        end
      for all descendants $y$ of $x$ such that
        ($x$.weight-$y$.weight+1)/$x$.weight $\in [1/S \cdots S]$ do
        begin
        ExciseDummy($x$,$y$)      {replace $y$ by DUMMY in
                                  $x$}
        FindStrategies($x$,Append($l$,$y$))
        FindStrategies($y$,$l$)
        TrySubstitutionSplit($x$,$l$,$y$)
        end
  end FindStrategies
  TryPLA, TryOperatorSplit, TrySubstitutionSplit:
  {These procedures calculate the dimensions of the layouts
    implied by their arguments. For the splits, all possible
    layouts resulting from combinations of strategies for the
    subparts are tried. The best strategies in various aspect
    ratio ranges are entered into the dictionary.}
  LookupStrategy($e$,$l$):
  {This function looks up in the dictionary the layout strat-
    egies for expression $e$ with excisions list $l$. Any members
    of $l$ which are not descendants of $e$, or are descendants
    of other members of $l$, are ignored. INIT is returned if
    no strategies have yet been sought for ($e$, $l$).}
```

V. Performance of the Regular Expression Compiler

The regular expression compiler has been implemented in *C* on a VAX/780. It can produce layouts using either the heuristic method or the dynamic programming method. By appropriately setting the parameters for the heuristic method, one can also find the layout as a single PLA or as a network of logic connecting individual symbol recognizers. This section will report how the compiler performs on some sample expressions.

The first series of expressions is the PR series. The PR2 expression was given in Section II. The others in the series have the same line and symbol declarations, and the following definitions (**any**^{n} is used as shorthand for n occurrences of **any**):

$$\begin{aligned} \mathrm{PR4} &= \mathbf{any}^2(\mathrm{PR2}) + \mathrm{PR2}\mathbf{any}^2 \\ \mathrm{PR8} &= \mathbf{any}^4(\mathrm{PR4}) + (\mathrm{PR4})\mathbf{any}^4 \\ \mathrm{PR16} &= \mathbf{any}^8(\mathrm{PR8}) + (\mathrm{PR8})\mathbf{any}^8 \\ \mathrm{PR32} &= \mathbf{any}^{16}(\mathrm{PR16}) + (\mathrm{PR16})\mathbf{any}^{16}. \end{aligned}$$

PRn is recognized whenever the last n inputs fail to match the first n. The results of running the regular expression compiler on the PR series is given in Table I. The times given in the last column are CPU seconds on the VAX. Areas are in $\lambda^2 \times 10^6$, where λ is half the minimum feature size (see Mead and Conway [7]). The "heuristic" results were the best that could be found by varying the parameters (there is another parameter, not shown, which indicates the desired shape of the final layout). It can be seen that both the heuristic method and the dynamic programming method are quite a bit better than the single-PLA or all-logic methods. Dynamic programming beats the heuristic method by an amount which increases with the expression size. Several dynamic programming results are shown to give some idea of the tradeoff between search thoroughness and execution time that occurs. Sketches of the layouts found by the compiler for PR16 are shown in Fig. 5(a) (heuristic) and (b) (dynamic programming). The boxes are the individual PLA's.

The next series of expressions to be tried were the SEQ expressions, where SEQn has the form

line l[n]
symbol $a1$(**l**[1]), $b1$(−**l**[1]), $a2$(**l**[2]), $b2$(−**l**[2]), $\cdots$, an(**l**[n]), bn(−**l**[n])
symbol any()
;
$b1$ + **any*** ($a1$ $b2$ + $a2$ $b3$ + $\cdots$ + an **any**++).

These expressions signal if the input wires are not turned on in sequence. The SEQ expressions are different from the PR ones in that they have a large number of input wires, so that the heuristic strategy (which does not pay attention to how many inputs a module needs) might be expected to do poorly. Another fact about these expressions is that the expression trees are tall and sparse. The PR expressions had rather bushy trees. Table II gives the results of using the regular expression compiler on the SEQ expressions.

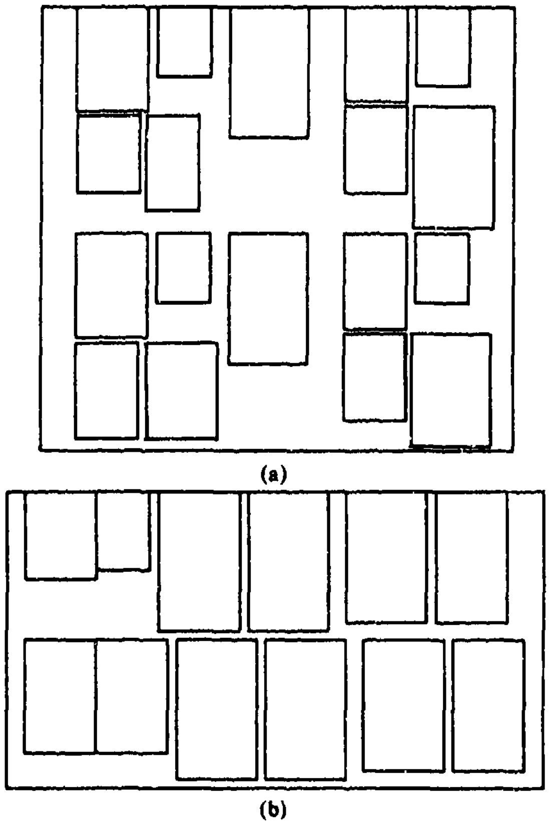
The final group of expressions is a slight modification of the SEQ group. To see what effect the depth of the tree has on the execution time, the BSEQ expressions were formed: they are just copies of the SEQ expressions without the $b1$+**any**++ at the beginning, factored so that they form completely balanced binary trees. For example, BSEQ4 is

(($a1$ $b2$ + $a2$ $b3$) + ($a3$ $b4$ + $a4$ **any**++)).

The results of compiling these expressions are also given in Table II. It can be seen that the compiler works faster on the bushy BSEQ expressions than it does on the corresponding SEQ expressions. This is because there are a smaller number of possible dummy nodes which satisfy the split-ratio requirement in the bushy trees.

TABLE I
DATA FOR PR EXPRESSIONS

Expression Name	Weight	Depth	Layout Method	L	H	S	Area (Mλ²)	Time (secs)
PR8	72	14	single PLA				.97	2.8
			all logic				.85	6.7
			heuristic	4	17		.58	2.8
			dyn. prog.	6	60	1.5	.54	14.0
			dyn. prog.	6	60	2.0	.55	24.0
			dyn. prog.	6	30	3.0	.55	55.7
PR16	160	23	single PLA				4.43	11.5
			all logic				2.28	15.3
			heuristic	4	17		1.69	6.0
			dyn. prog.	6	40	1.5	1.47	34.4
			dyn. prog.	6	30	2.0	1.23	159.6
PR32	352	40	single PLA				21.00	130.3
			all logic				8.88	35.9
			heuristic	4	17		3.87	17.3
			dyn. prog.	6	40	1.7	3.55	267.1
			dyn. prog.	7	25	2.0	3.19	1402.5


(a)

(b)

Fig. 5. Layout sketches for PR16. (a) Heuristic. (b) Dynamic programming.

VI. Evaluation and Conclusions

It has been shown that regular expressions have a structure which makes them quite amenable to a "divide-and-conquer" partitioning and placement procedure which runs fairly quickly. Clearly, the network-of-PLA's approach is superior to the single PLA or all-logic methods.

The program could certainly run a lot faster if substitution splits were not tried, but it has been found that these are definitely required. Perhaps the expressions could be parsed in such a way that the children would always be about the same weight: there is some freedom allowed because concatenation and union are associative operators. However, the closure operators form barriers to arbitrary reparsing, so in general one cannot balance the children.

The search over a range of possible dummy tree roots is

TABLE II
DATA FOR SEQ AND BSEQ EXPRESSIONS

Expression Name	Weight	Depth	Layout Method	L	H	S	Area (Mλ²)	Time (secs)
SEQ16	34	19	single PLA				.30	1.5
			all logic				.51	4.0
			heuristic	4	17		.28	2.1
			dyn. prog.	6	17	1.7	.24	5.0
SEQ32	66	35	single PLA				.97	3.5
			all logic				1.23	9.3
			heuristic	4	28		.84	3.4
			dyn. prog	6	70	1.7	.81	27.5
SEQ64	130	67	single PLA				3.48	9.2
			all logic				3.33	20.7
			heuristic	4	35		1.76	7.9
			dyn. prog.	6	30	1.7	1.62	168.0
BSEQ16	32	5	single PLA				.27	1.4
			all logic				.34	3.2
			heuristic	4	20		.33	1.6
			dyn. prog.	6	40	1.7	.23	2.7
BSEQ32	64	6	single PLA				.92	3.0
			all logic				.74	6.8
			heuristic	4	25		.59	3.6
			dyn. prog.	6	65	1.7	.59	8.9
BSEQ64	128	7	single PLA				3.39	9.8
			all logic				2.28	18.4
			heuristic	4	35		1.91	7.6
			dyn. prog.	6	30	1.7	1.53	15.9

another aspect which slows the compiler. If one tries only that node which yields the best weight ratio between the father and dummy trees, the resulting areas are somewhere between those found by the heuristic method and dynamic programming. For example, this modification led to the same layout as full dynamic programming for SEQ16, but for SEQ32 it only did as well as the heuristic method. It was found that one had to try the five best dummy tree roots before the full dynamic programming layout would be found for SEQ32. The execution times using the best-dummy-only modification were quite close to those of the heuristic method, so perhaps this is the most useful method of all, for small to medium sized expressions.

The dynamic programming method requires keeping a number of "best" layouts for expressions, in each of a number of different aspect ratio ranges. Varying the number of these ranges has some effect on the ability of the compiler to find good layouts. Originally, three ranges were used. This seemed to work, but when the compiler was changed to keep layouts for six ranges, the results were quite a lot better—at least for the larger expressions.

To sum up, each of the capabilities of the regular expression compiler adds incrementally to the quality of the layout, at a cost of extra execution time.

The work described in this paper has some resemblance to previous work on graph theoretic approaches to partitioning [10], but the problem is somewhat more tractable when trees are involved. Also, the idea of doing the placement by recursively splitting the plane into halves has been used before [6]. Not much has been done on automatically choosing a network of PLA's to implement a sequential circuit, although there has been some work done on optimizing single PLA's [9]. A circuit realization using a network of PLA's is given in [1], but the user must specify the splits with a hierarchical circuit definition.

The regular expression compiler is still undergoing improvements. Currently, the ability to have numerous "output signals" embedded in the expression is being incorporated. Also, more PLA optimizations are going to be done. In particular, nonoverlapping NFA states will be detected and a group of such states can be assigned binary-encoded state identifiers. This should reduce the current tendency for the PLA's to be fairly sparse. There are plans to use the compiler to generate much of the control logic for a VLSI chip being designed.

ACKNOWLEDGMENT

The regular expression compiler was originally designed and implemented by J. Ullman at Stanford University. The author has added the dynamic programming feature and made various other improvements.

REFERENCES

[1] R. Ayres, "Silicon compilation—A hierarchical use of PLA's," in *Proc. 16th Design Automat. Conf.*, June 1979, pp. 314-326.

[2] R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Tech. Rep. STAN-CS-80-798, Apr. 1980.

[3] M. J. Foster and H. T. Kung, "Recognizing regular languages with programmable building-blocks," in *VLSI-81*, J. P. Gray, Ed. New York: Academic, 1981, pp. 75-84.

[4] J. P. Gray, "Introduction to silicon compilation," in *Proc. 16th Design Automat. Conf.*, June 1979, pp. 305-306.

[5] D. Johannsen, "Bristle blocks, A silicon compiler," in *Proc. 16th Design Automat. Conf.*, June 1979, pp. 310-313.

[6] U. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation," in *Proc. 16th Design Automat. Conf.*, June 1979, pp. 1-10.

[7] C. Mead and L. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 1980.

[8] A. Mukhopadhyay, "Hardware algorithms for nonnumeric computation," *IEEE Trans. Comput.*, vol. C-28, pp. 384-393, June 1979.

[9] J. P. Roth, "Programmed logic array optimization," *IEEE Trans. Comput.*, vol. C-27, pp. 174-176, Feb. 1978.

[10] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electric circuits," in *Proc. 8th Design Automat. Workshop*, June 1972, pp. 56-62.

[11] D. P. Siewiorek and M. R. Barbacci, "The CMU RT-CAD system—An innovative approach to computer aided design," in *Proc. Fall Joint Comput. Conf.*, AFIPS, vol. 45, 1976.

[12] J. D. Williams, "STICKS—A graphical compiler for high level LSI design," in *Proc. Nat. Comput. Conf.*, 1978, pp. 289-295.

[13] G. Zimmerman, "Cost performance analysis and optimization of highly parallel computer structures: First results of a structured top-down design method," presented at 4th Int. Symp. on Comput. Hardware Description Languages, Oct. 1979.

Howard W. Trickey (S'79) received the B.A.Sc. degree in engineering science and the M.A.Sc. degree in electrical engineering from the University of Toronto, Toronto, Ont., Canada.

Currently, he is a graduate student in the Department of Computer Science, Stanford University, Stanford, CA. His technical interests include VLSI design automation and complexity theory, analysis of algorithms, and programming language design and implementation.

Mr. Trickey is a student member of the Association for Computing Machinery.

A Regular Expression Compiler*

by Howard W. Trickey and Jeffrey D. Ullman

Stanford University

Abstract

The regular expression notation can be used to specify some portions of a VLSI chip. These specifications are less procedural than the usual types of hardware languages, and this has advantages such as conciseness and understandability. We have implemented a compiler that takes a regular expression as input and produces a layout for a circuit that recognizes the expression. The input language is defined, and a portion of a computer controller is given to show how regular expressions can be used. We conclude by showing how the structure of regular expressions is an advantage when it comes to generating layouts.

Introduction

A silicon compiler is a program that takes a "high level" description of an integrated circuit and produces a layout for that circuit[2,5,12,14,15]. The reasons for using high level circuit descriptions are similar to those for using high level programming languages: the elimination of low-level errors, as well as the ease of writing, understanding, modifying, and verifying correctness of the descriptions. Overall, the use of silicon compilers is expected to help alleviate the high cost of VLSI design.

Broadly, there are two aspects to a circuit design. First, there is the *data path*: the registers, buses, adders, and other blocks for manipulating data. Then there is the logic used to control the data manipulation, which may be a few simple gates or a complicated microcode program. The data path part of the chip can be given a high level description by making heavy use of library cells, with various degrees of adaptability[8], or by using simplified layout diagrams[14]. At the moment, our compiler ignores this aspect of chip design. It is used only for generating the controlling logic portion.

Most approaches to control specification have a very procedural flavor. For example, actual programs for a register transfer level machine can be used to generate controlling logic[12,15]. A related approach is to give a finite state machine description that can be compiled into microcode[6]. One of the lessons of programming language compiler technology is that there are advantages to be gained by using a *data-driven* controller. Parsers and lexical scanners can easily be generated from succinct grammars[1]. Besides succinctness, these descriptions also have the advantage that they are static — one doesn't have to step through a program, remembering intermediate states, to understand what will happen. There is a school of thought that advocates organizing all programming tasks in a data-driven manner[7].

Our *Regular Expression Compiler* generates a layout for data-driven controller. Its input is a regular expression describing the input patterns expected, with embedded symbols that tell where outputs should be signaled. We allow more than one portion of a regular expression to be "active" at the same time. This makes it much easier to describe certain patterns; it can also be useful for allowing things to happen in parallel, with synchronization. With microcode there is usually only one locus of control, so that parallelism either has to be simulated (by interleaving), or multiple independent controllers must be used.

The Input Language

A regular expression is a notation for representing a set of strings of symbols. It is defined recur-

* Work supported by NSERC, NSF grant MCS-80-12907, and DARPA contract MDA 903-80-C-0107.

sively as follows:

- The symbol is the most basic kind of regular expression. In the application to circuits, the occurrence of a symbol means that the input wires must be zero or one, according to the symbol definition.

If E and F are regular expressions, then so are the following:

- The *union*, $E + F$, meaning either E or F.
- The *concatenation*, $E \cdot F$ (or simply EF), meaning E followed by F.
- The *closure*, E^*, meaning zero or more occurrences of E.
- The *positive closure*, E^{++}, meaning one or more occurrences of E.
- The *optional occurrence*, $E?$, meaning zero or one occurrence of E.
- (E), where parentheses are used for grouping. Unless parentheses are used, the unary operators have precedence over the binary operators, and concatenation has precedence over union.

The use of regular expressions to specify controllers is perhaps best seen by means of an example. The following is the complete input file required by the regular expression compiler for a small example:

```
line data[2]
symbol zero(data[1],-data[2]),
       one(-data[1],data[2]), any()
;
any (one any* zero + zero any* one) +
(one any* zero + zero any* one) any
```

The *line* declaration gives the wires that are input to the circuit. A line name can be subscripted (with [..]), as data is, to represent more than one wire. One can declare any number of lines. The *symbol* declaration gives the names of the symbols that will occur in the regular expression, with the values of the input wires which identify a symbol given in parentheses after its name. Here there are three symbols: zero, recognized when data[1] is a logical "1" and data[2] is a logical "0" (indicated by the "-" in front of data[2]); one, recognized when the data wires are reversed; and any, which doesn't specify either "1" or "0" for the data wires, so it is a "don't care." Note that any will be recognized at the same time as zero or one; there is no requirement that the wire combinations for different symbols be disjoint.

The regular expression itself follows the declaration. This expression gives all strings of symbols where either (a) the first symbol differs from the second-from-last symbol, or (b) the second symbol differs from the last symbol. This expression will be referred to as PR2.

The controller is a synchronous machine, acting as a pattern recognizer. The successive symbols of a string must appear in successive clock cycles for the pattern to be recognized. Whenever the symbols seen at preceding cycles form one of the complete strings specified by an expression, an output signal is given on a "recognized" wire for the expression.

Usually one wants to have a number of different output wires, signalled at various points during the recognition of a regular expression. This is done by first giving a declaration of the form output U,V (which declares U and V to be output symbols), and then embedding the output symbols at the desired points in the expression. For example, the regular expression

```
a b U c d V
```

where U and V are output symbols and the rest are input symbols says that after seeing a then b, emit output signal U; if the next two symbols are c then d, also emit V. No input cycle is spent expecting an "input" U.

The regular expression compiler also has a subexpression capability. One can declare something like subexp Wait = notready* ready, and then use Wait in another expression, as an abbreviation.

Regular expressions can specify highly parallel patterns such as PR2, or they can be used in a style that looks more like microcode. The following expression is a partial specification for a computer controller.

```
/* The ''main'' expression:
   instruction fetch-execute loop */
(InstructionFetch SourceFetch DestFetch Execute)*


/* SourceFetch subexpression:
   Load SOURCE register, according to mode */
```

```
subexp SourceFetch =
    sregdirect (SR-TO-BUS SOURCE-FROM-BUS)
    + (sregindirect (SR-TO-BUS MAR-FROM-BUS READSTART)
      + (sregautoincrement (SR-TO-BUS ALU-FROM-BUS
                                MAR-FROM-BUS READSTART)
            any (ALU-ADD ALU-CARRYIN)
            any (ALU-TO-BUS SR-FROM-BUS)
         + sregindexed (PC-TO-BUS ALU-FROM-BUS
                           MAR-FROM-BUS READSTART)
            any (ALU-ADD ALU-CARRYIN)
            any (ALU-TO-BUS PC-FROM-BUS)
            ReadWait MDR-TO-BUS ALU-FROM-BUS
            any (SR-TO-BUS ALU-ADD)
            any (ALU-TO-BUS MAR-FROM-BUS READSTART)
        )
        (indirect ReadWait MDR-TO-BUS
                               MAR-FROM-BUS READSTART)?
      ) ReadWait MDR-TO-BUS SOURCE-FROM-BUS

subexp ReadWait = readnotdone* readdone

. . .      /* Other subexpressions */
```

The lower-case symbols are inputs, the upper-case symbols are outputs, and the mixed-case symbols are subexpressions. The computer being specified has an architecture similar to the PDP-11. The hardware being controlled has a single bus attached to general registers, the ALU, SOURCE and DEST registers to hold instruction operands, and the memory address and memory data registers (MAR, MDR). The part of the specification reproduced above shows how the SOURCE register can be filled, according to the addressing mode. Most cycles start with a test of inputs that have been set according to the current instruction. Those that need no tests use a don't care input (any).

Layout of Regular Expression Recognizers

An advantage of using regular expressions to specify controllers is that their structure can be used as the basis for generating good layouts. It is quite easy to give a nondeterministic finite automaton that recognizes a regular expression, and to use a Programmable Logic Array (PLA) to simulate that automaton. Each state can be represented by a dynamic register whose value is calculated by the PLA using the inputs and the current state values (which are fed back from the registers). Details of this method are given by Floyd and Ullman[3].

Another possible implementation method is to use a module to recognize each symbol of the expression, and to combine the modules according to the structure of the expression, using appropriate logic gates. The methods of Foster and Kung[4] and Mukhopadhyay[10] lead to layouts similar to this. We use an intermediate approach, where the basic modules are PLA's, each of which is responsible for some subexpression of the total expression. Those modules are combined using logic and a "substitution" operation that allows one PLA to handle parts of the expression tree that have subexpressions removed. This turns out to be better than either of the extreme approaches, especially when dynamic programming is used to choose the best possible division into modules (see Trickey[13] for details).

One might wonder why we implement regular expressions as nondeterministic machines, rather than first converting them to deterministic machines, for which there exist standard reduction techniques. One reason is that the regular expression input exposes some important structure of the problem for which we are designing a circuit. We exploit this structure, for example, when we break the expression tree of a regular expression into subexpressions, each of which interacts with the others in very limited ways. If we converted to a deterministic finite automaton, it would be a large combinatorial problem to recover a partitioning of the states that was as good as what we can discover easily from the expression.

Another opportunity to use the structure of the input to the compiler comes when we do state assignment for the deterministic finite automaton that recognizes a given subexpression. We start with the nondeterministic finite automaton's states. If we simply identified each deterministic state with a subset of nondeterministic states, and identified each bit of the state code for deterministic states with a nondeterministic state, we would often have a poor encoding, since most subsets of the nondeterministic states would not be accessible, and the state encoding for the deterministic machine would have many more bits than is reasonable.

Fortunately, there is a better way to exploit the structure of the nondeterministic machine. We have developed a technique where we identify groups of nondeterministic states that do not *conflict*, in the sense that in any group at most one state can be active at any time. Then the states in each group can be binary encoded, essentially forming a deterministic machine for each group. This means that deterministic expressions do not suffer, since all the states will belong to one group, but at the same time we can handle highly parallel expressions (those where many subsets of the nondeterministic states are accessible). There are examples, including one we shall mention below, where we can achieve good encodings for deterministic machines of many more states than standard techniques could deal with.

As an example of our techniques, we have made measurements on the PRn series of expressions (PRn detects when the last n inputs differ from the first n, with "don't care" permitted). For PR8, both the single-PLA (naive encoding) and the module-per-symbol approaches yield layout areas of about 1 Megaλ^2. Our mixed approach combined with binary encoding gives a layout area of .35 Megaλ^2. A single PLA designed by hand, using best encodings we could think of, had an area of .42 Megaλ^2. Interestingly, our compiler will generate the same PLA if its parameters are fixed to force a single PLA. A deterministic finite state automaton to solve this problem would have about 2^{23} states, so it is clear that methods using deterministic machines could not be applied here.

Conclusions

We believe that the non-procedurality and succinctness of regular expression specifications for controllers is a definite advantage. They are easy to write, and they lend themselves to good implementations. A key property of regular expressions is that they expose the structure of the controllers. This structure, often buried in lower-level notations, can be exploited to find efficacious state assignments and partitions into PLA's. There are brute force methods to do this for deterministic machines, but they break down long before the regular expression methods.

We don't claim that we have as yet a complete set of facilities for creating real controllers — we are currently working with some large examples to see what else can be automated. A further possible direction is to combine our regular expression controllers with a data path design system.

References

[1] A.V. Aho, and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[2] R. Ayres. "Silicon Compilation — A Hierarchical Use of PLAs." *16th Design Automation Conf. Proceedings*, pp. 314-326, June 1979.

[3] R.W. Floyd, and J.D. Ullman. "The Compilation of Regular Expressions into Integrated Circuits." *Tech. Rep. STAN-CS-80-798*, Stanford Computer Science Dept., April 1980.

[4] M.J. Foster, and H.T. Kung. "PRA: Programmable Building Blocks for Recognizing Regular Languages in VLSI." *Unpublished memorandum*, Dept. of Computer Science, Carnegie-Mellon, 1981.

[5] J.P. Gray. "Introduction to Silicon Compilation." *16th Design Automation Conf. Proceedings*, pp. 305-306, June 1979.

[6] J. Hennessy. "SLIM: A Simulation and Implementation Language for VLSI Microcode." *Lambda*, pp. 20-28, Second Quarter, 1981.

[7] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[8] D. Johannsen. "Bristle Blocks, A Silicon Compiler." *16th Design Automation Conf. Proceedings*, pp. 310-313, June 1979.

[10] A. Mukhopadhyay. "Hardware Algorithms for Non-numeric Computation." *IEEE Transactions on Computers*, C-28, No. 6, pp. 384-393, June 1979.

[12] D.P. Siewiorek, M.R. Barbacci. "The CMU RT-CAD System — An Innovative Approach to Computer Aided Design." *AFIPS Fall Joint Computer Conference*, Vol. 45, 1976.

[13] H.W. Trickey. "Good Layouts for Pattern Recognizers." *Tech. Rep. STAN-CS-81-871*, Stanford Computer Science Dept., 1981.

[14] J.D. Williams. "STICKS — A Graphical Compiler for High Level LSI Design." *National Computer Conf. Proceedings*, pp. 289-295, 1978.

[15] G. Zimmerman. "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method." *4th International Symp. on Computer Hardware Description Languages*, October 1979.

September 1982

Report No. STAN-CS-82-927

Combining State Machines and Regular Expressions for Automatic Synthesis of VLSI Circuits

by

Jeffrey D. Ullman

Department of Computer Science

Stanford University
Stanford, CA 94305

COMBINING STATE MACHINES AND REGULAR EXPRESSIONS FOR AUTOMATIC SYNTHESIS OF VLSI CIRCUITS†

Jeffrey D. Ullman

Stanford Univ.

ABSTRACT

We discuss a system for translating regular expressions into logic equations or PLA's, with particular attention to how we can obtain both the benefits of regular expressions and state machines as input languages. An extended example of the method is given, and the results of our approach is compared with hand design; in this example we use less than twice the area of a hand-designed, machine optimized PLA.

I. The Regular Expression Compiler

A collection of routines have been written by H. Trickey and J. Ullman to translate regular expressions into circuits. At present, we first compile regular expressions into a language that describes nondeterministic finite automata (NFA's). These NFA's are then compiled into either PLA's or S. C. Johnson's *lgen* logic language.

A description of the regular expression language appears in [TU]. The language is quite standard, with perhaps the following exceptions.

1. Input symbols are not "disjoint," in the sense that at any time only one can be seen on the input. Rather, input symbols are defined in terms of some set of wires being on or off. Since not all wires must be specified for each symbol, there is the possibility that two or more symbols are on at a time. This has the consequence that apparently deterministic processes can in fact have nondeterminism in them.
2. Output signals are represented by ordinary-looking symbols in regular expressions. When the input is such that an output symbol is reached in the expression, we emit that signal, and proceed to recognize any continuation of the expression that the input allows us to recognize.

Example 1: In Fig. 1 we see an input to the regular expression compiler that forms a running example for this report. Without dealing now with the issue of what this program does, let us observe a few salient features. The first line says that there are seven input wires, $x[1], \ldots, x[7]$. Next come the definitions of the input symbols. For example, we see signal *in*0 on the input whenever the first wire is on and the second off. Note, for example, that we could see symbol *in*0 and also *acka*, if the first three wires were 1, 0, and 1.

† Work supported by DARPA contract MDA 903-80-C-0107.

Following this come the declaration of output signals, and then three subexpressions, *son.ein*, which is recognized when either *in*0 or *in*1 is seen, *waitin* which is recognized when neither input is seen, or both wires $x[1]$ and $x[2]$ are on simultaneously (which represents a "bad input," the symbol *badin*), and *allbut*01, which stands for the union of all signals but *in*0 and *in*1. After the declaration portion is a semicolon and the expression itself.

As an example of how the expression is to be interpreted, consider the seventh line of the expression, beginning *stateia*$\cdots$. It says that if we get a signal telling us we are in state *a*, and then receive any number of symbol *noacka* (*noacka** means "any number of *noacka*'s"), we emit the signal $OUTA$. We regard $x[3]$ as a wire that "acknowledges" the fact that signal $OUTA$ was received, so symbol *noacka*, defined by $x[3]$ being off, is seen until the *acka* symbol, $x[3] = 1$, is seen. In effect, we emit the output signal $OUTA$ until it is acknowledged.

When $OUTA$ is acknowledged by *acka* appearing on the input, the process of recognizing the expression proceeds to *waitin**, which is recognized for as long as the first two wires remain at 0, or both become 1 simultaneously (a bad input). Then, when *in*0 or *in*1 is seen, a signal to change to state *b* or *c* is made. If any of the symbols represented by *allbut*01 is received after the *acka*, an error is declared. □

II. Combining States and Expressions

The motivation for using regular expressions as a source language is twofold. First, they are a succinct and nonprocedural description of a large class of sequential processes. Thus they can provide some simplification in the design process for the right problem. Second, being structured descriptions of patterns, they are appropriate for proofs of correctness, and even if a formal proof is not attempted, they provide useful intuition that helps the reader convince himself of the correctness of the expression. In comparison, transition functions for automata are analogous to programs with goto's; they are inherently hard to understand and verify, either formally or intuitively.

On the other side of the coin, there are distinct advantages to process descriptions involving states. Often, it is natural to view a process as being in one of several states; for example, counting is especially easy when you have states available and very hard to do with regular expressions. It is the purpose of this report to descibe a simple modification to the regular expression compiler that allows us, in effect, to declare states and then define transitions among states in regular expression terms. As a result, we get the best of both worlds; states are available when they are more succinct than regular expressions or when they help us organize our design, and regular expressions are available when patterns of symbols are useful as a description of events.

```
line      x[7]
symbol    in0(x[1] -x[2])
          in1(x[2] -x[1])
          badin(x[1] x[2])
          acka(x[3])
          ackb(x[4])
          ackc(x[5])
          stateia(x[6] x[7])
          stateib(x[6] -x[7])
          stateic(-x[6] x[7])
          start(-x[6] -x[7])
          noin(-x[1] -x[2])
          noacka(-x[3])
          noackb(-x[4])
          noackc(-x[5])
output    OUTA
          OUTB
          OUTC
          stateoa
          stateob
          stateoc
          ERROR
subexp    somein=in0 + in1 + badin
subexp    waitin=noin + badin
subexp    allbut01=acka + ackb + ackc + badin
;
start waitin* (
          allbut01 ERROR +
          in0 stateoa +
          in1 stateob
          )
+
stateia noacka* OUTA (
                    (ackb+ackc+somein) ERROR +
                    acka waitin* (
                              allbut01 ERROR +
                              in0 stateob +
                              in1 stateoc
                              )
                    )
+
stateib noackb* OUTB (
                    (acka+ackc+somein) ERROR +
                    ackb waitin* (
                              allbut01 ERROR +
                              in0 stateoc +
                              in1 stateoa
                              )
                    )
+
stateic noackc* OUTC (
                    (acka+ackb+somein) ERROR +
                    ackc waitin* (
                              allbut01 ERROR +
                              in0 stateoa +
                              in1 stateob
                              )
                    )
```

Fig. 1. Input to regular expression compiler.

To introduce states into the regular expression language, we make the following modifications.

1. The names *stateiX* for any X are input symbols that represent the fact that we have just entered state X. Symbol *start* serves as the initial state. To indicate that these states are disjoint, i.e., we can be in only one of them at a time, we can use imaginary wires, such as $x[6]$ and $x[7]$ in Fig. 1, to make it appear to the compiler that at most one of these input symbols can be present on the input at any time. Of course, if the states were not disjoint in this sense, we could express the legal subsets by another combination of dummy wires.
2. Output symbols *stateoX* for any X are used as goto's. If we emit the symbol *stateoX*, we shall in effect turn on the input symbol *stateiX* and enter state X.

The complete regular expression consists of the sum of expressions that begin *start* and *stateiX* for the various X's. The portion of the regular expression associated with each state is recognized, if possible, each time we enter that state, and we make whatever outputs the regular expression tells us to make in response to what inputs we see.

After the regular expression compiler converts the expression into a nondeterministic finite automaton, an edit script is used to identify the input symbol *stateiX* with the output symbol *stateoX* and make certain other changes so things work properly.

Example 2: A case in point is the problem to which the regular expression program of Fig. 1 is a solution. This program implements the transmitter from [AUY] that sends bits reliably over a channel that has a high probability of losing bits, but does not change 0's into 1's or vice-versa. This view of a channel is plausible if we assume that any noise or other error causes the system to fail to detect a bit. This strategy, of assuming no signal whenever something goes wrong, is modeled after the Datakit protocol [F].

The general idea is that when the transmitter is given a bit to send, it sends one of three signals *OUTA*, *OUTB*, or *OUTC*, chosen by a method to be described. It keeps sending the signal until it receives an acknowledgement of the signal sent. Then, it stops sending the signal until the next input, 0 or 1, is received, whereupon it sends the next signal (in the sense that *c* follows *b*, which follows *a*, which follows *c*) if 0 is input, and it sends the previous signal in this cyclic order if 1 is input.

The purpose of this arrangement is so that whenever the transmitter sends a new input, it changes the signal sent; that change serves to acknowledge the acknowledgement. If we did not always make a signal change, the receiver could not tell, say upon receiving two 0's, whether these were two different inputs, or the acknowledgement of the first had been lost, and the second 0 was a retransmission of the first.

Another way to look at the signal selection algorithm, is that we count one for an input 0 and two for an input 1, and transmit *OUTA*, *OUTB*, or *OUTC* depending on whether the sum of inputs received so far is congruent to 1, 2, or 0, modulo 3. Counting, even counting modulo 3, is very difficult to express in the regular expression language. Thus it is natural to introduce three states, *a*, *b*, and *c*, that are entered whenever we receive an input that makes this running modular sum 1, 2, or 0, respectively.

We already discussed briefly in Example 1 what happens in one of these states, say *a*. After receiving the *stateia* signal to say we have entered state *a*, we emit *OUTA* for as long as the input matches *noacka**, that is, the *acka* acknowledgement is not received. A sequence of *noacka*'s can be followed by either of two events that cause special outputs. First, the *acka* signal can be received, and then, after any sequence of *waitin*'s, i.e., no input, an *in*0 or *in*1 triggers a signal that causes a jump to another state, *b* or *c*, respectively. After receiving *acka*, any input but *in*0 or *in*1 causes an error signal. Note that *allbut*01 and *waitin* can be seen at the same time, so we can continue waiting for a good input even while signaling when errors occur.

Now let us return to the point in the expression where we are recognizing *noacka** and waiting for *acka*. At the same time we are waiting for *acka*, if we receive *ackb*, *ackc*, or *somein*, we have an error condition; in the first two cases, the wrong acknowledgement was received, in the last, we received either a bad input, or a good input before we are ready to transmit it. In this case, we emit the output signal *ERROR*. Note that all of these error conditions are seen on the input at the same time *noacka* is seen, so emitting *ERROR* does not prevent us from continuing to see *noacka** and eventually to see *acka* and another input. However, inputs received erroneously do not cause a change of state, because we cannot reach a term like *in*0 *stateob* in the regular expression until after the *acka* has been received.

The portions of the expression following *stateib* and *stateic* are analogous to what we have described. The portion following *start* differs only in that we are not waiting for an acknowledgement, and if any is received it is an error.

The result of compiling Fig. 1 is shown in Fig. 2. This figure illustrates the NFA language used. Each type of statement begins with a unique letter. For example, D is a declaration of an input symbol, much like in the regular expression compiler. However, note that the input symbol *stateiX* and the output symbol *stateoX* have both become *stateX*, and this input symbol is declared (in lines 6–8, e.g.) to be present when the wire of the same name is on; that wire is the corresponding output signal.

The letter N indicates the name of the NFA, and F indicates the name of the final signal, if any (there is none in Fig. 2), and the states that cause the final signal to be emitted. Letter I introduces the name of the initial signal, *init* in this case, and a list of the initial states, *st*2, *st*3, and so on.

A state is declared by the letter S, followed by the state name, and the input symbol that it recognizes. The NFA language is restricted in that each state recognizes only one input symbol. However, this restriction is not bothersome for NFA's that are output by the regular expression compiler, and in general, we can create

```
D in0 ( in0 -in1)
D in1 ( -in0 in1)
D badin ( in0 in1)
D acka ( acka)
D ackb ( ackb)
D ackc ( ackc)
D statea (statea)
D stateb (stateb)
D statec (statec)
D noin ( -in0 -in1)
D noacka ( -acka)
D noackb ( -ackb)
D noackc ( -ackc)
N nfa1
F ;
I init; st2 st3 st4 st5 st6 st9 st11
S st2 noin
T st2 st3 st4 st5 st6 st9 st11
S st3 badin
T st2 st3 st4 st5 st6 st8 st9 st11
S st4 acka
T st8
S st5 ackb
T st8
S st6 ackc
T st8
S st7 O stateb
S st8 O error
S st9 in0
T st10
S st10 O statea
S st11 in1
T st7
S st12 O statec
S st13 statea X
T st5 st6 st14 st15 st17 st18 st19 st22
S st14 noacka
T st5 st6 st14 st15 st17 st18 st19 st22
S st15 O outa
S st16 badin
T st4 st5 st6 st8 st9 st11 st16 st29
S st17 badin
T st8
S st18 in0
T st8
S st19 in1
T st8
S st20 badin
T st4 st5 st6 st17 st20 st27 st31 st33
S st21 statec X
T st4 st5 st17 st18 st19 st25 st26 st37
S st22 acka
T st4 st5 st6 st17 st23 st24 st30 st32
S st23 noin
T st4 st5 st6 st17 st23 st24 st30 st32
S st24 badin
T st4 st5 st6 st17 st23 st24 st30 st32
```

Fig. 2(a). Beginning of NFA description.

```
S st25 acke
T st4 st5 st6 st9 st11 st16 st29
S st26 noacke
T st4 st5 st17 st18 st19 st25 st26 st37
S st27 noin
T st4 st5 st6 st17 st20 st27 st31 st33
S st28 ackb
T st4 st5 st6 st17 st20 st27 st31 st33
S st29 noin
T st4 st5 st6 st9 st11 st16 st29
S st30 in0
T st7
S st31 in0
T st12
S st32 in1
T st12
S st33 in1
T st10
S st34 stateb X
T st4 st6 st17 st18 st19 st28 st35 st36
S st35 noackb
T st4 st6 st17 st18 st19 st28 st35 st36
S st36 O outb
S st37 O outc
C st2; st3 st4 st5 st6 st8 st9 st11
C st3; st4 st5 st6 st8 st9 st11
C st4; st5 st6 st8 st9 st11 st16 st17 st18 st19 st20 st23 st24 st25 st26 st27 st28 st29 st30 st31 st32 st33
     st35 st36 st37
C st5; st6 st8 st9 st11 st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st25 st26 st27 st29 st30 st31
     st32 st33 st37
C st6; st8 st9 st11 st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st27 st28 st29 st30 st31 st32 st33 st35 st36
C st7; st8
C st8; st9 st10 st11 st12 st14 st15 st16 st17 st18 st19 st20 st22 st23 st24 st25 st26 st27 st28 st29
     st30 st31 st32 st33 st35 st36 st37
C st9; st11 st16 st29
C st11; st16 st29
C st13; st21 st34
C st14; st15 st17 st18 st19 st22
C st15; st17 st18 st19 st22
C st16; st29
C st17; st18 st19 st20 st22 st23 st24 st25 st26 st27 st28 st30 st31 st32 st33 st35 st36 st37
C st18; st19 st22 st25 st26 st28 st35 st36 st37
C st19; st22 st25 st26 st28 st35 st36 st37
C st20; st27 st31 st33
C st21; st34
C st23; st24 st30 st32
C st24; st30 st32
C st25; st26 st37
C st26; st37
C st27; st31 st33
C st28; st35 st36
C st30; st32
C st31; st33
C st35; st36
E
```

Fig. 2(b). End of NFA description.

several states with the same predecessors to simulate one state with transitions on several inputs.†

An O preceding the symbol associated with a state means that the symbol is an output symbol, rather than an input symbol. The letter X following the symbol, as in *st*13, means that the state is *external*; it is always on and waiting to see its input symbol.‡ It is exactly the states of the NFA that represent the states used in the regular expression specification that become external states of the NFA.

All states are followed by the letter T and a list of their transitions, that is, their successors. Finally there are conflict statements introduced by the letter C. The state following the C is declared to conflict with all the states after the semicolon. *Conflicting states* are those that can be on at the same time, a result not only of the nondeterminism but of the fact that several input symbols may be recognizable at once. Conflicts among states are taken into account when we find a coding of the NFA's states for an implementation. □

III. Logic Generation

The NFA is converted to the logic language *lgen* by an algorithm described in [U]. Briefly, the nondeterministic states must receive representations that will enable us to identify that each state is on, regardless of what other states are also on at the same time. Here is where knowing the state conflicts may help, because when state *i* is on, that fact can only be obscured by states that conflict with *i* also being on. For example, if the NFA were really deterministic, there would be no conflicts, and we could use a binary coding of the states.

One way to code states is to give each a private signal. Then we can tell the state is on independent of any other states. The actual approach taken by the logic generator used is slightly more sophisticated. It attempts to identify *groups*, which are sets of mutually nonconflicting states.†† Within a group, binary codes are selected so that any conflicting states from other groups will receive the same code. To do so, a minimal number of states that would make this coding impossible are expelled from groups and given private signals. Groups of a single state are similarly given private signals.

The result is that in addition to private signals, there are *code bits* and *group bits*. A state without a private signal is recognized by the bit of its group being on, and the code bits being on or off as appropriate

† [T] describes a more general NFA language that allows, multiple transitions, ϵ-transitions, and a variety of options not available in the NFA language described here.

‡ There is another kind of state like external states, that does not appear in Fig. 2. These states, called advance states and designated by A, are like external states, but when they see their input, they enable their successors to recognize their own inputs at the same time unit. Advance states are needed to implement correctly networks of NFA's that together recognize one large regular expression. Large expressions need to be broken into pieces implemented by separate NFA's for two reasons. First, processing large expressions is too time consuming, especially minimizing the states of the NFA and computing conflicts. Second, the circuits implementing the NFA's such as PLA's or Weinberger arrays, will be too large and badly shaped if the NFA has too many states.

†† However, before looking at conflicts, states that have exactly the same predecessors (and therefore are really just different transitions from the "same" state) are combined into one.

```
ost2 == ¬in1 * ¬in0 * est2
ost3 == in1 * in0 * est2
ost4 == acka * est4
ost5 == ackb * est5
ost6 == ackc * est6
ost9 == ¬in1 * in0 * est9
ost11 == in1 * ¬in0 * est9
ost13 == statea
ost14 == ¬acka * ce1 * ¬ce2 * ¬ce3
ost17 == in1 * in0 * est17
ost18 == ¬in1 * in0 * est18
ost19 == in1 * ¬in0 * est18
ost22 == acka * ce1 * ¬ce2 * ¬ce3
ost16 == in1 * in0 * ¬ce1 * ce2 * ¬ce3
ost29 == ¬in1 * ¬in0 * ¬ce1 * ce2 * ¬ce3
ost20 == in1 * in0 * ce1 * ce2 * ¬ce3
ost27 == ¬in1 * ¬in0 * ce1 * ce2 * ¬ce3
ost31 == ¬in1 * in0 * ce1 * ce2 * ¬ce3
ost33 == in1 * ¬in0 * ce1 * ce2 * ¬ce3
ost21 == statec
ost25 == ackc * ¬ce1 * ¬ce2 * ce3
ost26 == ¬ackc * ¬ce1 * ¬ce2 * ce3
ost23 == ¬in1 * ¬in0 * ce1 * ¬ce2 * ce3
ost24 == in1 * in0 * ce1 * ¬ce2 * ce3
ost30 == ¬in1 * in0 * ce1 * ¬ce2 * ce3
ost32 == in1 * ¬in0 * ce1 * ¬ce2 * ce3
ost28 == ackb * ¬ce1 * ce2 * ce3
ost34 == stateb
ost35 == ¬ackb * ¬ce1 * ce2 * ce3
est2 == LAST fst2 CLEAR globalinit + init
fst2 == ost2 + ost3
est4 == LAST fst4 CLEAR globalinit + init
fst4 == ost2 + ost3 + ost22 + ost16 + ost29 + ost20 + ost27 + ost21 + ost25 + ost26 + ost23 + ost24 +
     ost28 + ost34 + ost35
est5 == LAST fst5 CLEAR globalinit + init
fst5 == ost2 + ost3 + ost13 + ost14 + ost22 + ost16 + ost29 + ost20 + ost27 + ost21 + ost25 + ost26 +
     ost23 + ost24 + ost28
est6 == LAST fst6 CLEAR globalinit + init
fst6 == ost2 + ost3 + ost13 + ost14 + ost22 + ost16 + ost29 + ost20 + ost27 + ost25 + ost23 + ost24 +
     ost28 + ost34 + ost35
est9 == LAST fst9 CLEAR globalinit + init
fst9 == ost2 + ost3 + ost16 + ost29 + ost25
est17 == LAST fst17 CLEAR globalinit
fst17 == ost13 + ost14 + ost22 + ost20 + [illegible] + ost21 + ost26 + ost23 + ost24 + ost28 + ost34 + ost35
est18 == LAST fst18 CLEAR globalinit
fst18 == ost13 + ost14 + ost21 + ost26 + ost34 + ost35
error == ost3 + ost4 + ost5 + ost6 + ost17 + ost18 + ost19 + ost16
stateb == ost11 + ost30
statea == ost9 + ost33
statec == ost31 + ost32
outa == ost13 + ost14
outc == ost21 + ost26
outb == ost34 + ost35
ce1 == LAST cf1 CLEAR globalinit
cf1 == ost13 + ost14 + ost22 + ost20 + ost27 + ost23 + ost24 + ost28
ce2 == LAST cf2 CLEAR globalinit
cf2 == ost16 + ost29 + ost20 + ost27 + ost25 + ost28 + ost34 + ost35
ce3 == LAST cf3 CLEAR globalinit
cf3 == ost22 + ost21 + ost26 + ost23 + ost24 + ost34 + ost35
```

Fig. 3. Logic implementing communication protocol.

to its code.

Example 3: In Fig. 3 we see the output of the NFA-to-logic compiler, with certain header information, indicating clocking and the borders on which signals appear, omitted. Because it turns out that there is only one nontrivial group, and that group does not have exactly a power of two states, we were able to eliminate the group bit, and, by not using the all-zeros code for any state in the group, detect the presence of a state in the group by one or more of the code bits being on.

The overall organization of the logic in Fig. 3 is not unlike that of a PLA. The variables are in three groups, designated by the letters e, f, and o. The first group, e, corresponds to columns in the and-plane of a PLA and represents the fact that a certain state is "enabled"; if its input symbol is now seen it can enable its successors for the next input cycle. Some states have private enablers, like *est*2 for state 2. Other states are coded, and in Fig. 3 there are three coded enabler variables *ce*1, *ce*2, and *ce*3, combinations of which represent the enablers for various states. Note that not every state has an enabler, either private or coded. States without enablers have the same entering transitions as some other state that does have an enabler, and the same enabler serves for both.

The f group of variables are "feedback"; they correspond to columns in the or-plane. State fX at one time unit becomes eX at the next time unit by means of *lgen* statements such as

$$est2 = \text{LAST } fst2 \text{ CLEAR globalinit} + init$$

which says that state 2 is enabled either by the initial signal *init*, or by *fst*2 being on at the previous time unit. The output signals, such as *statea* or $OUTA$, also correspond to columns of the or-plane.

The o group of variables correspond to the terms of the PLA. For each state there is an o variable that is turned on when the state is enabled, and the proper input is seen. For example, line 1 of Fig. 3 says *ost*2 is on whenever state 2 is enabled (*est*2 is on) and *noin* is seen on the input (detected by both wires *in*0 and *in*1 being off). Line 9 says that *ost*14 is turned on when input *noacka* is seen and state 14 is enabled (represented by the coded enabler bits being 100).

The only difference between a PLA structure and the organization of the variables in Fig. 3 is that the *statea*, *stateb* and *statec* variables do not fit into the scheme. Rather, we can view them as fed back from the or-plane, where they are generated, to the and-plane, where they are used, with no delay due to clocking. Thus, Fig. 3 can be used almost directly as input to a PLA generator that permits unclocked signals as an option.

IV. Evaluation of Results

It is difficult to compare the logic of Fig. 3 with the "best possible" logical description of an equivalent circuit. It appears that, when the ability of the *lgen* compiler to eliminate common subexpressions and perform other optimizations is taken into account, the resulting logic will be very close to that found in the hand designed PLA described below. Thus, we are optimistic that our automatic synthesis method behaves very well when amount of logic generated is the criterion used.

We can obtain a more concrete estimate of the quality of the circuit designed if we view it as a PLA specification and compare it with a PLA designed carefully by hand. In our hand design, we used three feedback wires. Two were used to binary code the "state," i.e., whether we were in the start condition, or in what we have called states *a*, *b*, and *c*. The third feedback wire indicated whether we were waiting for an acknowlegement or had received the acknowledgement and were waiting for the next input. Terms based on this encoding were written down and optimized using the *gry* PLA optimizer [H]. The resulting PLA had:

1. 32 terms.
2. 17 columns in the and-plane, representing an initializing signal, the three feedback wires, and the five input wires (*in*0, *in*1, *acka*, *ackb*, and *ackc*), each of which except the initializer requires inversion.
3. 7 columns in the or-plane, representing the three feedback wires and four outputs (*ERROR*, *OUTA*, *OUTB*, and *OUTC*).

The resulting area is 32 * (17 + 7) = 768.

In comparison, the PLA constructed directly from Fig. 3 requires the following:

1. 30 terms (the o variables plus one term to carry the initial signal to the or-plane).†
2. 27 columns in the and-plane, consisting of
   a) 10 columns for the inputs and their complements.
   b) 7 columns for the private state enablers; these do not have to be inverted.
   c) 3 columns for *statea*, *stateb*, and *statec*; these also do not require inversion.
   d) 6 columns for the three coded enablers, which do require inversion.
   e) 1 column for the initial signal.
3. 17 columns in the or-plane, consisting of four output signals and 13 feedback wires.

The resulting size is 30 * (27 + 17) = 1320. This figure is 72% greater than the hand-designed one. The overhead of the PLA borders will tend to reduce this figure somewhat, as will the fact that clocking is not needed on six of the columns of the machine-generated PLA. But the fact that space is required for 13

† It is not unusual for PLA's generated from regular expressions to have fewer terms than hand-designed ones, because the former PLA's tend to use one-hot codes (private enablers) for states, and that sort of code costs columns, but may save terms.

```
line    x[5]
symbol  in0(x[1] -x[2])
        in1(x[2] -x[1])
        badin(x[1] x[2])
        ack(x[3])
        stateia(x[4] x[5])
        stateib(x[4] -x[5])
        stateic(-x[4] x[5])
        start(-x[4] -x[5])
        noin(-x[1] -x[2])
        noack(-x[3])
output  OUTA
        OUTB
        OUTC
        stateoa
        stateob
        stateoc
        ERROR
subexp  somein==in0 + in1 + badin
subexp  waitin==noin + badin
subexp  allbut01==ack + badin
;
start waitin* (
        allbut01 ERROR +
        in0 stateoa +
        in1 stateob
        )
+
stateia noack* OUTA (
                somein ERROR +
                ack waitin* (
                        allbut01 ERROR +
                        in0 stateob +
                        in1 stateoc
                        )
                )
+
stateib noack* OUTB (
                somein ERROR +
                ack waitin* (
                        allbut01 ERROR +
                        in0 stateoc +
                        in1 stateoa
                        )
                )
+
stateic noack* OUTC (
                somein ERROR +
                ack waitin* (
                        allbut01 ERROR +
                        in0 stateoa +
                        in1 stateob
                        )
                )
```

Fig. 4. Revised input to regular expression compiler.

feedback wires for the machine-generated PLA will serve to increase the ratio.

V. Correction of Errors

One important advantage of the regular expression approach to design, as with high-level descriptions in general, is that modifications are easier to make, and make reliably, than with ad-hoc designs.

Example 4: It turns out that our design of Fig. 1 is not the simplest that meets the specifications of [AUY]. Rather, since the channel is assumed never to make a mutation error, only to lose signals, there is no need to distinguish between the three acknowledgements; whenever we receive an acknowledgement, we know it was actually sent by the receiver, and we know that if the receiver is not broken, then it was sent in response to the receipt of the correct signal.

Thus, we should modify Fig. 1 in the following two ways.

1. *acka*, *ackb*, and *ackc* should be identified as the signal *ack*; similarly, their complements, *noacka*, etc., are identified as *noack*.
2. While waiting for an acknowledgement, there are no "wrong acknowledgements" to receive, so terms like

$$(ackb + ackc + somein)\ ERROR$$

should be replaced by

$$somein\ ERROR$$

The resulting revision is shown in Fig. 4.

VI. Conclusions

Regular expressions, in conjunction with conventional state machine definitions of processes, is a promising way to design circuits at a very high level. The use of optimizing logic compilers to do the actual implementation may be superior to PLA implementation of regular expressions, but the evidence of the case described in this paper, and other cases we have analyzed by hand, is that even PLA implementation of regular expressions comes within a factor of two of the area used by hand-designed PLA's. Further, the problem of coding nondeterministic states is not yet fully resolved, and there is hope that better PLA and/or logic implementations of regular expressions will be developed in the future.

References

[AUY] Aho, A. V., J. D. Ullman, and M. Yannakakis, "Modeling communications protocols by automata," *Proc. Twentieth Annual IEEE Symposium on Foundations of Computer Science*, pp. 267–273, 1979.

[F] Fraser, A. G., "Datakit—a modular network for synchronous and asynchronous traffic," *Proc. IEEE Intl. Conf. on Communications*, 1979.

[H] Hemachandra, L. A., "GRY, a PLA minimizer," unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.

[T] Trickey, H., "Regular expressions and NFAs," unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.

[TU] Trickey, H. and J. D. Ullman, "A regular expression compiler," *Proc. IEEE COMPCON*, 1982.

[U] Ullman, J. D., "Description of NFA-to-logic compiler," unpublished memorandum, Dept. of C. S., Stanford Univ., 1982.

ICTEST:
A Unified System for Functional Testing and Simulation of Digital ICs

J. M. Watson,
J. A. Newkirk,
R. Mathews,
and D. B. Boyle

Information Systems Laboratory
Stanford University
Stanford CA 94305

Abstract

ICTEST is an algorithmic language for describing functional tests of digital integrated circuits. The test stimulus and response specification is high-level, with the ICTEST system handling the translation into a test vector. The ICTEST system unifies testing and simulation: the same test program may be compiled to run against any of the 2 simulators and 3 testers in the system. The ICTEST system has been operational for over 1½ years, has been used by over 80 Stanford student and staff designers, and has proved to be an effective tool for functional testing in a fast-turnaround environment, where cutting testing time and cost is critical.

Introduction

Since the fall of 1979, when introductory LSI design classes based on [1] were first offered[†] at Stanford, there has been an explosive growth in LSI design activity, both in classes and for research purposes. We realized at the outset that a convenient functional-test system would be required if a substantial number of the resulting chips were going to be tested — breadboarding individual test fixtures would be a daunting task. We developed the ICTEST system in response, and it has since seen heavy use by the Stanford community.

The ICTEST System

The idea behind the system structure (Fig. 1) is to unify testing and simulation. The designer writes one test description that can be compiled for any of the functional simulators and testers in the system. Currently, tests target to either of 2 simulators or 3 testers (the *esim* and *tsim* switch simulators [2]), 2 bench-top testers developed locally, and a Tektronix S-3260 system).

The heart of the system is ICTEST itself. It provides a uniform front-end language for specifying stimulus and response, and the compiler and runtime system provide interfaces to each target. All of the targets are available on-line, with errors and results returned immediately in a standard format. The user is thereby insulated from knowing the details of each target.

An important benefit of this structure is that the same test is used to exercise both the simulation and the fabricated IC. The process of debugging the test program against the simulator contributes to design verification, since both the stimulus and response are specified in the test program. Many design errors that cause differences between specifications and circuit behavior will show up at this stage. Test integration is simplified because the test program has been validated, so errors encountered during testing are typically the result of fabrication defects.

An alternative approach to generating the prospective tester responses would be to run

is work was supported by DoD ARPA under contracts MDA-903-79-C-0680 .d MDA-903-80-0432.

[†] Taught by Drs. Newkirk and Mathews.

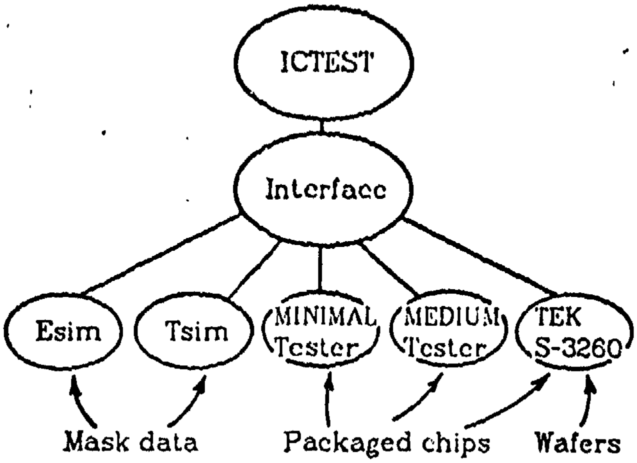


Figure 1. The ICTEST system. ICTEST itself, the interface programs, and the *esim* and *tsim* switch-level simulators run on a VAX-11/780. The VAX communicates with the testers over serial links. The MINIMAL tester can test 40-pin devices at a maximum of 1000 drive or sense operations per second; the MEDIUM tester, 80 pins at 100,000 operations per second; the TEK tester, 64 pins at 5,000,000 operations per second, all pins simultaneously.

the stimulus vector against the simulator and to use the simulator output as the expected response for testing. However, this approach fails to validate that the simulated output is correct. It is not nearly as comprehensive as requiring that 3 different descriptions of the design's behavior — the ICTEST response specification, the simulation result, and the fabricated part's output — coincide exactly.

The ICTEST Language

ICTEST is an embedding of testing features in C [3]. (C is an Algol-like language, similar to PASCAL.) These features divide into three major classes: port declarations, specifying the inputs and outputs of the design; test statements, for operating on ports; and control constructs, for specifying concurrency and pipelining. The full power of C is available for describing tests algorithmically. To elaborate these points, we shall present an example.

Consider a serial, 2's-complement multiplier [4], as shown in Fig. 2. The multiplier accepts two N-bit operands, least-significant bit first, and (because of the recoding and computation used) produces the product beginning $3*N/2$ clock cycles later. The product is scaled so that the numerical result is:

$$Z = X*Y*2^{-2(N/2-1)}$$

Additionally, the multiplier is pipelined. As soon as the input of one pair of operands is complete, the next pair may be applied, even though the first product has not yet emerged.

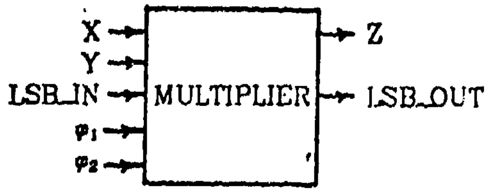


Figure 2. Serial multiplier

A representative ICTEST program for the multiplier appears in Fig. 3. The bond statement simply specifies the pinout of the part. The port declarations specify the clocking and formatting associated with each logical input or output. For example:

```
input X serial lsb N valid phi1;
```

indicates that X accepts N-bit numbers serially, least-significant-bit first. Each bit must be valid around the falling edge of the φ_1 clock.

The procedure *mult*() tests a single product. The driving (=) and sensing (=?) operations on the ports are self-explanatory; note that ICTEST, not the user, is responsible for formatting the operands and the product correctly. The brackets "[[" and "]]" indicate that all of the enclosed operations are to start concurrently, *e.g.*, the operand values are driven out, starting on the same clock cycle. The notation "(3*N/2)::" indicates that the product should be sensed beginning $3*N/2$ clock cycles later.

The main program *test*() requests two sequential ("{{" and "}}") multiply tests, 3*4 and 7*7. The **pack** keyword instructs ICTEST to pack the two tests together as tightly as possible without causing collisions on the ports. Without packing, the two tests would take $3*N/2+N$ clock cycles each; with packing, the tests are pipelined, and the second test begins N cycles after the first, for a total of $3*N/2+2N$ cycles.

```
#ifdef SIM /* if this is simulation */
#include mult_bond.sim
#else
/* else we're testing actual chip */
bond pkg 14 socket 14
          /* chip and tester socket
           * both have 14 pins */
  {
/* bond ports to pin numbers */
          X=1;
          Y=2;
          LSB_IN=3;
          Z=6;
          LSB_OUT=5;
          Mainclock=8,9;
  }
#endif

#define N 6 /* precision */

/* port declarations follow */
input X serial lsb N valid phi1;
input Y serial lsb N valid phi1;
input LSB_IN serial lsb N valid phi1;
output Z serial lsb N valid phi2;
output LSB_OUT serial lsb N valid phi2;
clock Mainclock;
/* test procedures */
test()
  {
     Mainclock pack{{
                          mult(3,4);
                          mult(7,7);
                      }}
  }

mult (a,b)
int a,b;
  {
    [[/* start concurrent context */
                  LSB_IN = 1;
                  X = a;
                  Y = b;
         (3*N/2):: LSB_OUT =? 1;
         (3*N/2):: Z =?
             floor (a*b*exp(2, (-2(N/2-1))));
    ]]
}
```

Figure 3. A small test of the serial multiplier

The example shows how the features of ICTEST contribute to a concise and easily understood description of a test. An important final step is to relate detected errors back to the test description. The error display produced by ICTEST includes the port name, the expected and actual values, and the line number in the test program source where the sense operation was requested. This information is sufficient for the user to isolate the source of the problem by referring to the ICTEST source program, without having to analyze the actual test vector.

Related Developments

The ICTEST system is a unique, practical synthesis of functional testing ideas. Unlike ATLAS [5], it is intended expressly for functional testing of digital ICs; ATLAS is extremely clumsy for this purpose. [6] is concerned only with high-level simulation. [7] and [8] contain languages that are probably suitable for a unified test/simulation system, although neither drives a simulator. Neither language is as high-level as ICTEST, however, in formatting capabilities or concurrency specification. Similarly, ANGEL [9] lacks the ability to describe serial input/output and pipelined or concurrent activity. ANGEL can generate tests for multiple targets, but testing is not on-line and errors are not referenced to the source program.

Languages provided by commercial test manufacturers (*e.g.*, see [10]) are both lower level and much more complex, in part because they are more general-purpose than ICTEST; they consequently demand considerable expertise to use. Front-end macro languages ease the situation somewhat, but tests remain laborious to construct.

Practical Experience with ICTEST

Researchers and students in classes at Stanford have used ICTEST since February of 1981. Since then, over 80 users have tested about as many designs using the ICTEST system. The principal targets have been the *esim* simulator and the MEDIUM tester. The idea of using the simulator target for debugging test programs has worked very well.

An informal survey of ICTEST users reveals that some users prefer an interpretive simulator interface for debugging their design: a major objection to ICTEST is its compilation time. However, users with designs well matched to ICTEST capabilities – *e.g.*, pipelined architectures, circuits with serial inputs or outputs, circuits performing mathematical functions, or circuits that perform concurrent functions – uniformly prefer ICTEST. The behavior of such chips is very difficult to describe without the facilities that ICTEST provides.

Some circuits have been tested with the Tektronix S-3260 tester. This task would have been completely unmanageable if were it not for ICTEST. Student designers were able to test their circuits at speed without learning any of the details of the S-3260 or its programs.

Conclusion

The ICTEST system is a practical, effective tool for functionally simulating and testing digital ICs. Such perennial problems as serial [illegible], concurrency, and pipelining are audressed by language features. Unifying simulation and testing simplifies debugging of test programs and permits validated test suites to be developed before the design is sent for fabrication.

The ICTEST system has been operational for over 1½ years and has been used by many student and staff designers. Experience has shown that ICTEST is effective for functional testing in a fast-turnaround environment, where testing must be easily accessible to a wide range of designers.

References

1. Mead, C. A., and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980
2. Baker, C. M. and Terman, C., "Tools for Verifying Integrated Circuit Designs," *Lambda*, Fourth Quarter, 1980.
3. Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, New Jersey, 1978.
4. Lyon, R. F., "Two's Complement Pipeline Multipliers," *IEEE Trans. Comm.*, April, 1976, pp. 418-425.
5. *ATLAS Test Language*, IEEE Std. 416-1981, IEEE, New York, 1981.
6. Hill, D. D., "Language and Environment for Multi-Level Simulation," SEL Tech. Report No. 185, Stanford University, 1980.
7. Pasco, R., "Smalltalk IcTester User's Manual," Xerox Inter-Office Memorandum, PARC-SSL-LSI, 1980.
8. DeBenedictis, E. P., "FIFI Test System Preliminary User's Manual," ARPA Contract Report No. 4270, 1981.
9. Snoulten, B. and Peacock, J., "ANGEL — Algorithmic Pattern Generation System," Proc. 1981 Intl. Test Conference, IEEE Comp. Soc. Press, 1981.
10. *S-3260 Automated Test System Reference Manual*, Tektronix, Inc., P.O. Box 500, Beaverton, Oregon 97077, November, 1977.

Dumbo, A Schematic-to-Layout Compiler

Wayne Wolf, John Newkirk, Robert Mathews and Robert Dutton

Center for Integrated Systems

Stanford University

Abstract

This paper describes a technique for translating logic circuit descriptions into layouts and a program, Dumbo, to demonstrate the technique. The technique allows the designer to transform any digital circuit into an arbitrary layout. The designer describes the circuit in terms of logical primitives to determine the function and physical primitives to describe the form of the cell's layout. Dumbo synthesizes a placement and wiring for the components, generating a stick diagram that can be compacted into a layout. The compiler chooses default physical structures if none are specified by the designer; this allows the designer to vary the effort expended on a cell according to its importance. Implementation of Dumbo has focused on nMOS technology. Experimental results show that the resulting system gives good results for a large class of cells with much less effort on the designer's part than required for traditional design techniques.

Introduction

The advantages of a VLSI design methodology that emphasizes a structured approach to both the logical and physical design [1] are clear. In order to make such an approach feasible for large chips, we must allow the designers to push detail onto subordinates, preferably computers, so that they can concentrate on problems critical to the design. This paper describes a designer's assistant, Dumbo, that compiles circuits and abstract physical structures into layouts.

We conjectured that it would be possible to compile stick diagrams from simple descriptions because not all cells are equally hard to design. Many cells in a chip are not very area-critical and some not at all. These cells tend to be straightforward applications of common physical structures to implement logical elements. Abstractions of these structures can be used to describe the desired cell, letting the compiler worry about their details.

To test this hypothesis we built a cell compiler, Dumbo. Our implementation works with nMOS circuits, but the techniques used are not restricted to that technology. Dumbo takes as input a connectivity list of the circuit and a description of its external connections. It then solves a placement and routing problem to

*This research was supported by ARO contract DAAG29-80-K-0046 and by DARPA contract MDA903-79-C-0680

synthesize a planar topology (topography) for the cell and wires it. Dumbo's output is a stick diagram that can be turned into a layout by a sticks compactor; Figure 1 shows a compiled stick diagram and its compacted layout.

The compilation process is extremely flexible: if physical structures are not given for parts of the circuit the compiler will use simple structures as defaults. If the result is inadequate, the designer can improve the compiled cell as necessary by adding structural detail to the input. In this way the designer can match the effort in designing the cell to the cell's importance in the overall design.

A stick diagram compiler is a logical next step given the existence of sticks compactors, and the compaction problem is well understood. Some typical sticks compactors are SLIM [2] and FLOSS [3], which are shearing compactors, and CABBAGE [4] and Lava [5], which are critical path compactors. Lava was used as the back end compactor in this work.

The topography synthesis problem has attracted much attention in the past several years. Related work includes single-layer block layout [6], automatic generation of schematic artwork [7] [8], one-dimensional gate layout, including Weinberger's work [9] [10] [11], and the miniature gate array cells produced by SLAP [12].

The next step in automatic layout synthesis is full topography synthesis, which is the subject of this paper. To describe our solution, we first enumerate the goals for Dumbo. We then describe the algorithms used to design cells and give the results of quality measurements on Dumbo-designed cells.

Goals

Given our emphasis on structural plus logical description of cells, we decided that these properties were necessary for a practical cell design system:

1. The program should be able to work with arbitrary digital logic circuits, including pass transistor designs. (We do not insist that parasitic circuit elements--capacitance, resistance--be precisely controllable.)

2. Cells should not be forced into a limited set of topographies. The program should be able to synthesize arbitrary layouts.

3. The program should design an electrically correct layout from an absolute minimum of information.

4. The program should be controllable and predictable: the algorithms should be able to use information in addition to the minimal description, and the designer should be able to add information and correctly predict the result.

5. Execution time should be no more than a few CPU minutes. This restriction limits the extent of

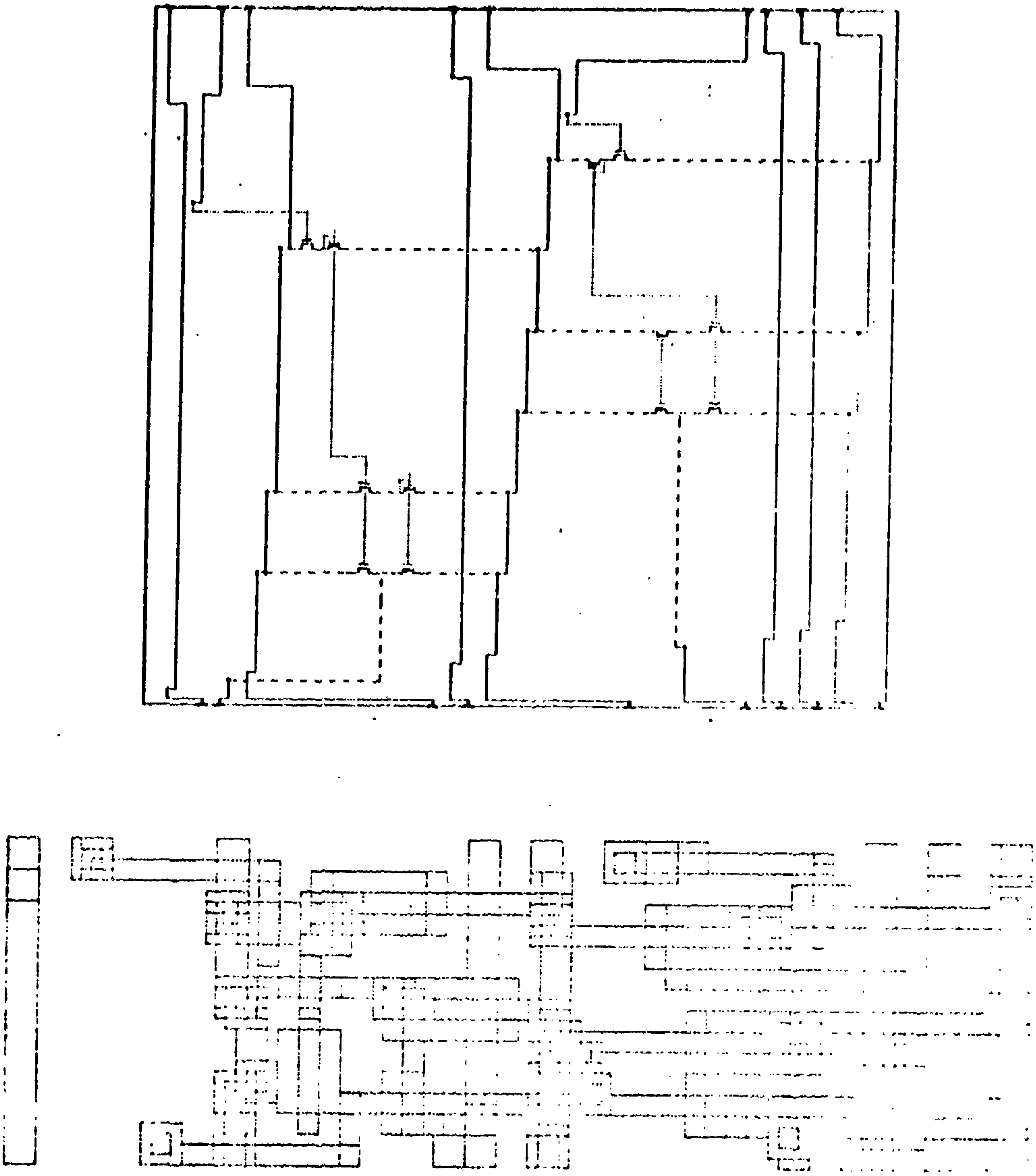


Figure 1: An automatically generated stick diagram and its layout

the program's search. We will trade a slightly more detailed input description for computation time in order to achieve area-efficient results.

The Compilation Process

The schematic-to-layout process occurs in two steps: first Dumbo produces a stick diagram; then the stick diagram is translated into a layout by a sticks compactor. Dumbo itself is implemented as a pipeline of five programs (see Figure 2). Each program adds detail to the description until a unique stick diagram results. The state of a simple cell at various stages in the pipeline is shown in Figure 3.

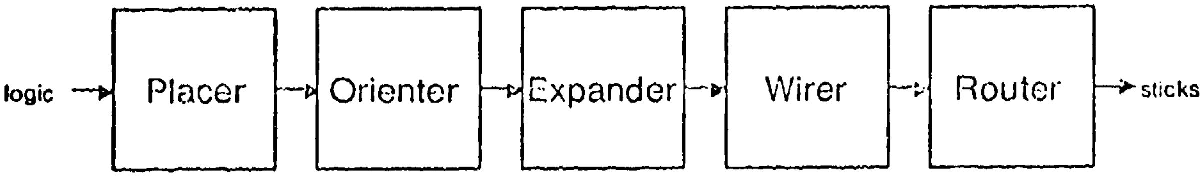
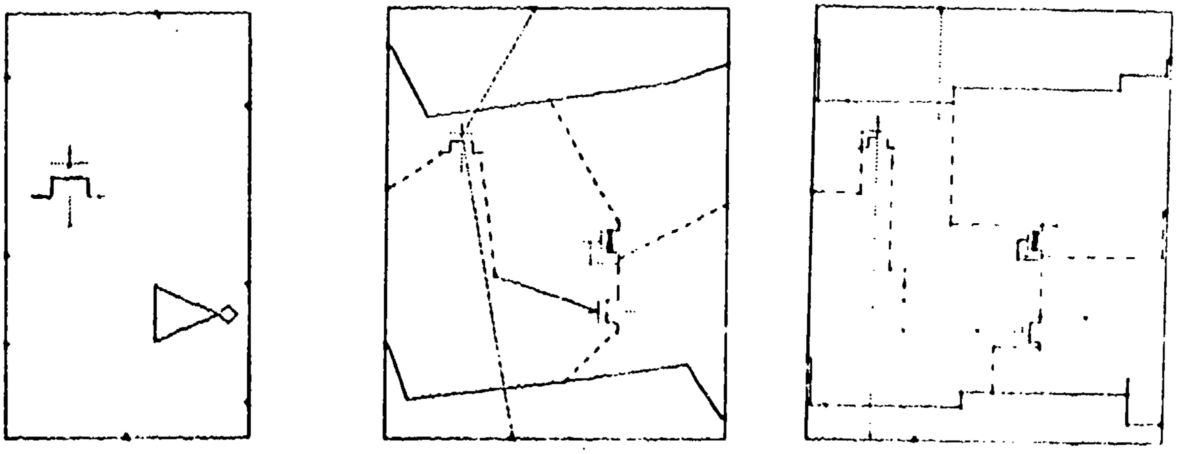


Figure 2: The Compilation Pipeline

Briefly, the programs and their functions are: the placer, which establishes the relative positions of gates, transistors and external pins; the orienter, which gives orientation and mirroring to components requiring orientation; the expander, which translates functional blocks, such as inverters, into transistor circuits that implement the functions; the wirer, which breaks nets into trees of pairwise connections or branches; and the router, which defines the precise path that each branch will take.



After Placement and Orientation

After Expansion and Wiring

After Routing

Figure 3: Partial Designs of a Cell

The pipeline can be partitioned roughly into the same functional blocks as the typical channel router:

placement of components (placer, orienter and expander); wiring tree generation or loose routing (wirer); and final routing (router). While this analogy is useful, it is important to remember that the channel routing and leaf cell problems, and the methods used to solve them, are very different. The channel routing problem works with components (blocks) that are very much larger than the wires connecting them. In the leaf cell synthesis problem, on the other hand, components and wires are comparable in size.

The Input Description

The input to Dumbo has two sections: an electrical description of the circuit, and structural information suggesting the desired layout. The level of detail in the electrical description is fixed. In contrast, the amount of structural information given depends on the degree to which the designer wants to control the form of the output. It is by varying the size and content of the structural description that the designer matches the cell to the requirements.

Electrical description

The electrical description lists the components in the circuit and the connections among these components. Each electrical node (net) is described by a list of the component pins connected to the net. A component can be an external pin, a transistor or a compound component made up of several transistors. Compound components are used to describe common functional blocks. Figure 4 shows two compound components, an inverting superbuffer and a multiplexer.

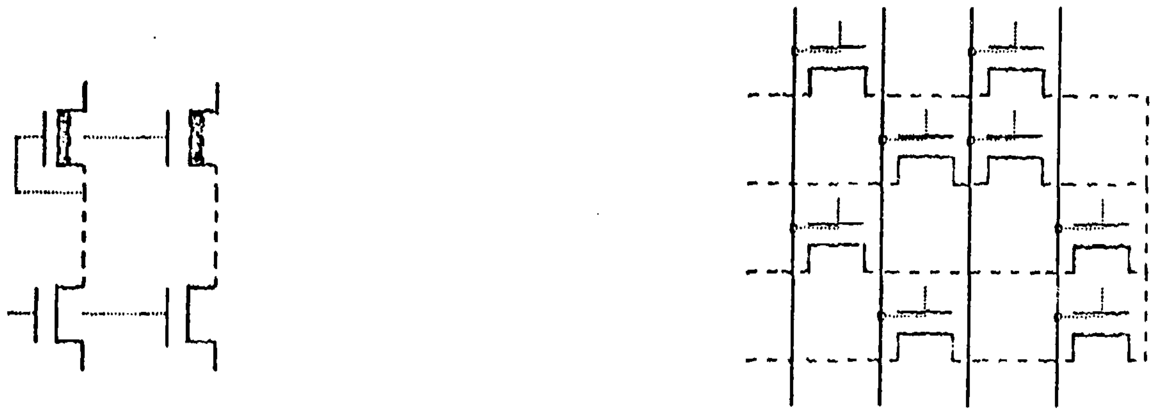


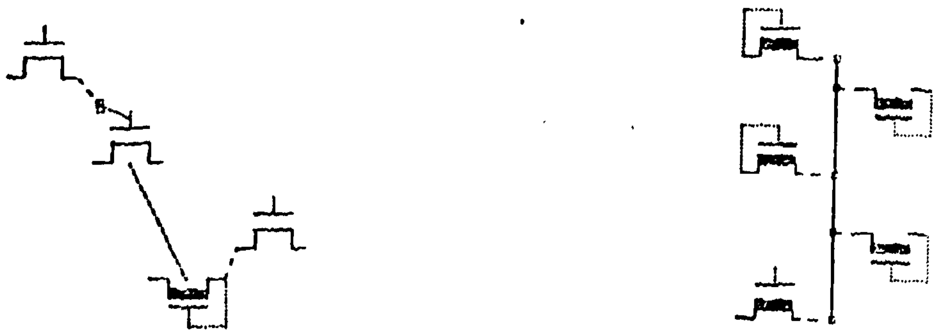
Figure 4: Compound components

Structural Description

The minimum structural information required is a description of the external connections into the cell. Each pin is placed on one of four cell edges--north, south, east or west--and the pins on each edge are partially ordered. The pin ordering is required because it defines the cell's communication and helps define the cell structure; it is a partial ordering so that the placement process can help define the communication plan.

The designer can optionally specify additional structural information about both nets and components. These structures specify the form that the layout is to take; the compiler fills in the details.

There are two wiring structures, the spanning tree and the track. Examples of these primitives are shown in Figure 5.



Spanning Tree *Track*

Figure 5: Wiring structures

If the pins on the net being wired are considered as nodes on a net, then the spanning tree is a set of arcs that directly connect all the nodes. Tracks differ in that connections can also be made indirectly through intermediate nodes that are added to the problem. These intermediate nodes are roughly collinear and the wires between them form the spine of the track. Spurs connect the points on the spine to the component pins that are to be connected. Tracks are modeled after the long sets of control lines commonly used in layout.

A net can be implemented as a single wiring structure, or it can be composed of several structures each wiring a subset of the pins on the net. For instance, a net may be wired as two separate tracks connected by a spanning tree. Thus the designer can build arbitrary structures for a net from combinations of the basic wiring structures.

Compound components are used to supply structural information, in addition to their role in the electrical description. Functional units usually have one or a few best layouts. The best layout for a function may vary with the application, but generally a particular layout for a function is used many times in a design. The designer chooses a topography for a function by specifying a template to be used. Each template has a different placement of the internals and a different set of prewired connections. Templates supply information on the relative placement of elements of the function and on device sizing; the internal electrical connections can be made by the template designer or by the wiring program in any combination. Figure 6 shows two templates for an inverter that differ in the placement of their internal transistors.

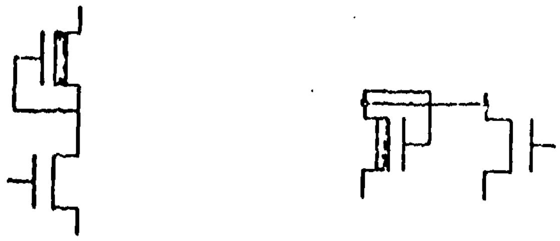


Figure 6: Component Templates

The final method of structural specification is to give implementation hints to the programs in the pipeline. The hints mechanism is discussed in the next section.

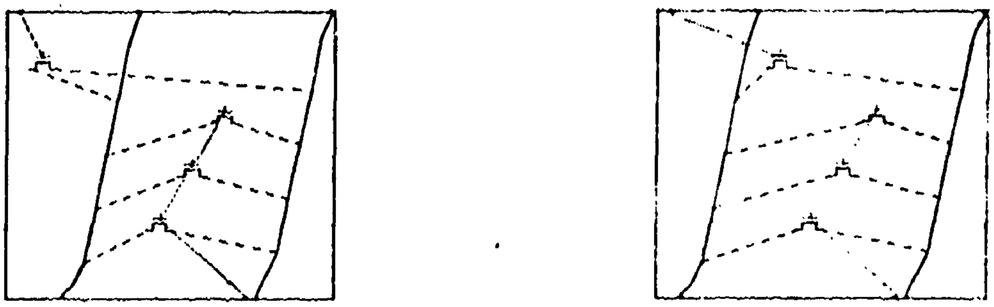
Algorithm Summary

We now discuss the algorithms used to implement each phase of the compiler. The phases will take as input, in addition to the information described above, hints that specify a part of the solution. The final solution will be in general changed by the existence of the solution seed supplied by the hint. For instance, the wiring phase will take as a hint a branch to be included in the spanning tree of a net; the existence of this first branch may be enough to produce a different spanning tree. Hints allow the designer finer control of the cell design than do structural elements; a hint is a rough equivalent to a block of assembly language code inserted into a high-level language program. While hints are not intended for extensive use, they are useful and add to the controllability of the compiler. Each algorithm summary below describes the hints that the algorithm will recognize.

Placer

The placer produces two total orderings of the transistors, logic elements and external pins in the cell, one in x and one in y. The algorithm used is a modified form of the force-directed placement method of Quinn and Breuer [13]. The force-directed model uses springs to guide placement. For each net a complete graph of springs is built so that every pair of pins on the net is connected by a spring. The system is then relaxed to find the minimum energy state, under the restriction that external pins are allowed to move in only one dimension along the cell edge. The ordering of components in the stable state is taken as the placement.

Hints take the form of added springs. The user specifies a pair of components that are to be attached by a spring and the stiffness of the attached spring. Figure 7 shows how user-defined springs may modify the placement. The original placement put the topmost transistor to the left of both tracks connected to its source and drain. To move it between the tracks the designer stiffened the springs between the source and drain and the two tracks.



Before

After

Figure 7: Placement Hints

Orienter

The orienter uses a torque-directed algorithm to compute the orientation of each component. Consider the component to be oriented as a pivot point around which rotate a set of arms, one for each component pin; then a wire connecting to a pin exerts a torque on the arm that drives the component to a preferred orientation, as shown in Figure 8.

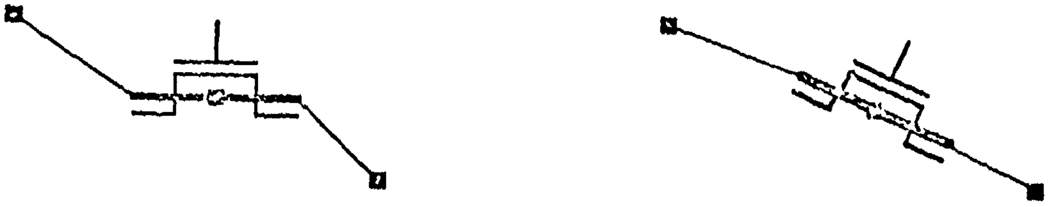


Figure 8: Torque-directed Orientation

The torques exerted by each connection to the component can be summed to find the torque that rotates the component to its minimum energy orientation. Since there may be two stable states, one with wires crossed and one with uncrossed wires, mirroring may also be necessary. Mirroring is done so that the minimum number of wires cross over the component to get to their destination. The solution is actually calculated by a simple voting algorithm. Each branch votes on the orientations it prefers and the orientation with the most votes wins. If the orientation is not satisfactory the user can override it with a hint.

Expander

The expander converts a description of compound components and transistors into a description containing only transistors. The template specified by the designer for each compound component is used as a macro for the transistor implementation of the function; if no template is specified a default template is used. The template contains information on device sizes and on placement of internal components, which may be transistors or vias. It may also specify parts of the internal wiring of the function. For this stage the hint supplied by the designer is the template to be used for an instance of a compound component.

Wirer

The wirer translates each net into a set of pairwise connections (branches) with assigned layers and introduces vias where they are required. Two algorithms are used, one for spanning trees and the other for tracks. The spanning tree algorithm finds the minimum spanning tree of a complete graph of branches, which contains all connections on the net. Branches to be included in the tree can be supplied by the designer as hints, or by earlier stages of the compiler, and they are guaranteed to be in the tree. Vias are inserted into the cell on a branch-by-branch basis: a via is put into every branch that connects pins on different layers.

Track nets are implemented as a particular form of Steiner tree. The spine of the net is built at a location specified in the placement, and spurs connect the Steiner points on the spine to components on the net. The Steiner points can be simple points or vias, depending on the layers of the spine and the component connection.

Router

The router translates the branches created by the wirer into a Manhattan stick diagram, one whose line segments all intersect at right angles. The cell routing problem is more general than the channel routing problem. In a channel routing problem the channels are formed by the four edges of the blocks that are being wired; but in the cell routing problem the transistors to be connected can be considered as point objects, so there are no natural wiring channels.

Our area routing algorithm, called L-routing, uses a very simple form of interconnect: all branches are drawn with two line segments in the form of either an L or an inverted L (see Figure 9).

If the problem is constructed such that no two pins share a common centerline then no branches will be left

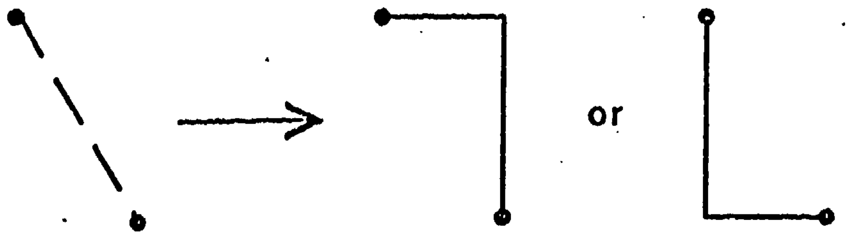


Figure 9: L-routing

unrouted: all undesired shorts can be removed by adding a jumper to a shorted branch that temporarily changes the wire to a safe layer. The routing problem is therefore to find a set of orientations for the Ls that minimizes the number of shorts. The optimal L-routing problem is NP-complete. However, typical cells do not require an exhaustive search to find a reasonable solution. A straightforward rip-up algorithm is sufficient to give good results. The algorithm also allows the designer to specify a routing for a branch as a hint.

Measurements

It would be desirable to test Dumbo's effectiveness by directly comparing compiled stick diagrams to hand-drawn stick diagrams. However, a stick diagram is only a means toward the desired end of a manufacturable layout, and there is no reasonable quality measures for stick diagrams. Therefore, Dumbo's performance must be measured through the filter of a sticks compactor. Figure 10 shows our method for measuring the quality of compiled layouts.

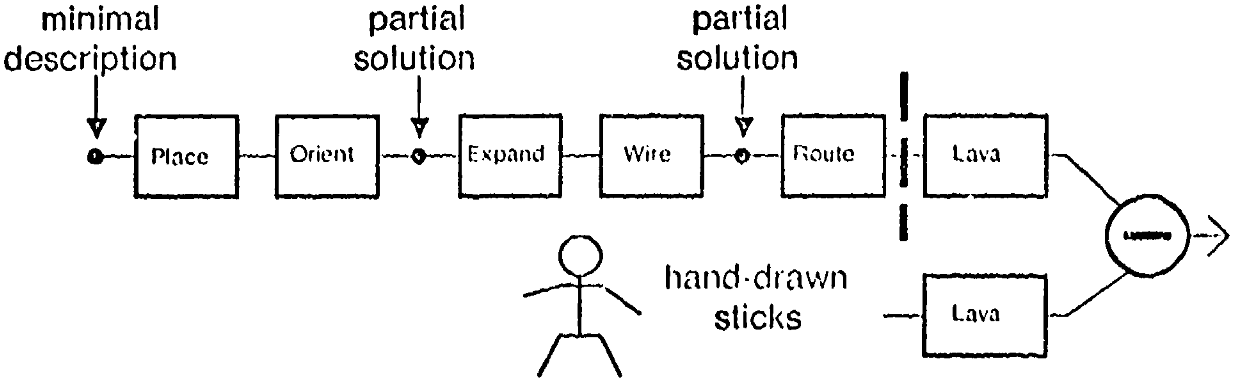


Figure 10: Comparing hand-drawn stick diagrams to compiled sticks

First, a test cell is rendered as a stick diagram by hand and compacted. Then a Dumbo description of the cell is written and compiled into a stick diagram, and that stick diagram is also compacted. Finally the areas of the two layouts are compared.

We can also measure the layout quality as a function of the work expended on the input. We do this by inserting a partially complete description of the cell into the middle of the compilation pipeline. For instance, to measure the penalty incurred with automatic placement we prepare a description of the cell that has the same placement and orientation of the components as does the hand-drawn stick diagram, and we start compilation of this description at the expander. We have taken measurements starting at three stages of the pipeline: starting at the placer, which gives a totally automatic cell; starting at the expander, which is equivalent to generating a layout from a schematic drawing (taking the component placement from the drawing); and starting at the router, which measures routing efficiency.**

The cells used for comparison are taken from actual designs. Primary sources are Stanford chip design projects and the Stanford Cell Library [14]. Seven cells have been tested to date; they range in complexity from six to 24 transistors and cover a variety of physical design styles.

Starting stage	Area penalty (% larger than hand design)
placer	120%
expander	60%
router	15%

Table 1: Average area penalty compared to hand-drawn sticks

Table 1 summarizes the area penalties incurred by Dumbo, rounded to 1.5 significant digits. The variance of area penalty for designs introduced at the router is small. The variance for designs introduced at later stages is larger. Area penalty is not directly related to final cell size, cell aspect ratio or number of components. The area penalty for automatically placed cells is caused by the force-directed placement algorithm's lack of concern with the planarity of wiring trees, forcing the router to add jumpers to the cell. The minimum energy placement is in general not the placement that gives the fewest intersecting wires. The wirer adds area to a cell by choosing a bad location for a via; since the wirer does not search for the best place to add a via it may cross two branches when need not have occurred. (Crossed branches increase the chance that a jumper is added during routing.) The results for area penalty starting from the expander are skewed by two very thin cells that suffered from poor via placement. Since the hand-drawn cells are very thin any

**This strategy of evaluating stick diagrams by examining their compacted layouts assumes that the compactor is equally good at compacting hand drawn and machine-drawn stick diagrams. This is in fact not the case. Current compaction algorithms are sensitive to the geometry of the input problem and experienced sticks designers take this problem into account, while Dumbo does not. We have therefore started work on a more robust compaction algorithm and this algorithm was used to prepare our results.

increase in width incurs a large area penalty. If these cells are thrown out the average penalty for compilation starting at the expander is about 30%. Our plans for improvements to the routing and wiring algorithms are outlined in the conclusion.

Area penalty is most strongly related to the number of jumpers added during routing; the number of jumpers required depends on other properties of the cell topography. A single jumper adds two vias to the component count. These extra components and the required spacing around them incur a large penalty in cell area. In order to understand this penalty we correlated the area penalty to the number of jumper vias added during routing (see Figure 11). The percentage of components added is calculated as the number of jumper vias added divided by the number of transistors and vias input to the router.

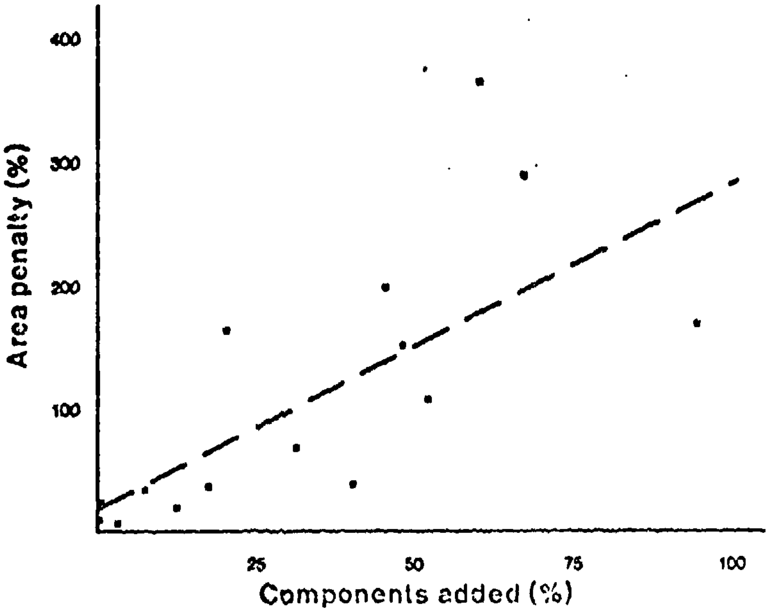
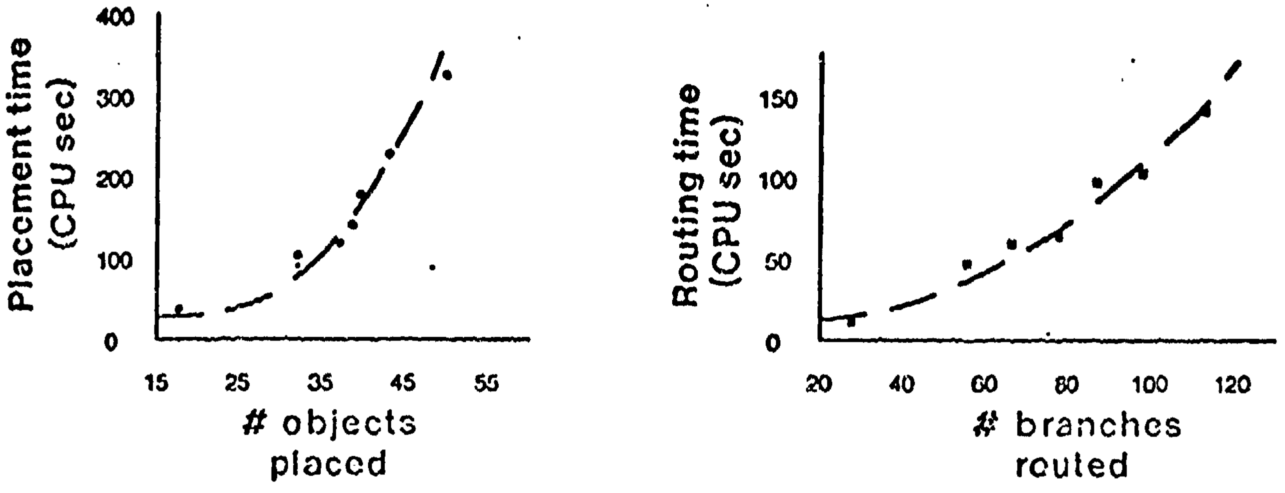


Figure 11: Increase in component count versus area penalty

The percentage increase in area of a cell is roughly 2.5 times the increase in component count caused by jumpering. The area penalty per jumper is large because jumpers add unusable white space to the cell as well as the area of their vias. A jumper via standing between two wires will leave a strip of white space the width of the via (plus design rule spacing) next to the via and between the wires.

The vast majority of the CPU time required to run Dumbo is spent in placement and routing. Figure 12 shows the average time complexity of these programs; in the placement time graph transistors, compound components, external pins and tracks are counted as placed components. Both programs show a quadratic relationship between input complexity and computation time. The placement algorithm's performance is



Placement

Routing

Figure 12: Computation Time vs. Cell Complexity

limited by the matrix operations performed; sparse matrix techniques are of no help since the coefficient matrices are not sparse. The speed of the router is limited by the method used to sort branches. The time complexity of routing could be reduced to O(n log n) from O(n^2) if more care were taken in implementation.

The measurements were made on a VAX/11-780 with floating point acceleration running Berkeley Unix[***] . The Dumbo programs were written in Pascal and the compactor was written in C.

Conclusion

We have presented in this paper a method for compilation of leaf cell layouts from simple, abstract descriptions. Cell layouts share a set of physical primitives that can be succinctly described and reconstructed using straightforward techniques. These compilation techniques allow the designer to vary the design effort expended on a cell, with a commensurate payoff in output layout quality. These properties of the compiler encourage the designer to concentrate on the most critical areas of the design, so the designer can be more productive.

Our future work will concentrate on improvements in the compilation techniques used, notably the placement and wiring algorithms. We expect to be able to greatly enhance the placer with the addition of an improvement phase that concentrates on maximizing the planarity of the wiring. We are also working on a wiring algorithm that searches for good locations for vias. These enhancements should significantly improve the quality of compiled cells.

[***] Unix is a Trademark of Bell Laboratories.

References

1. Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1979.

2. A. E. Dunlop, "SLIM-The Translation of Symbolic Layouts into Mask Data," *Journal of Digital Systems*, Vol. V, No. 4, 1981, pp. 429-451.

3. Y. E. Cho, A. J. Korenjak and D. E. Stockton, "FLOSS: An Approach to Automated Layout for High-Volume Designs," *14th Design Automation Conference Proceedings*, ACM/IEEE, 1977, pp. 138-141.

4. Min-Yu Hsueh, *Symbolic Layout and Compaction of Integrated Circuits*, PhD dissertation, University of California, Berkeley, December 1979.

5. Robert Mathews, John Newkirk and Peter Eichenberger, "A Target Language for Silicon Compilers," *Compcon Proceedings*, IEEE Computer Society, Spring 1982, pp. 349-353.

6. S.B. Akers, J.M. Geyer and D.L. Roberts, "IC Mask Layout With a Single Conductor Layer," *Seventh Design Automation Workshop Proceedings*, ACM/IEEE, 1970, pp. 7-16.

7. James A. Smith, *Automated Generation of Logic Diagrams*, PhD dissertation, University of Waterloo, 1975.

8. R. J. Brennan, "An Algorithm for Automatic Line Routing on Schematic Drawings," *Design Automation Conference Proceedings*, ACM/IEEE, 1975, pp. 324-330.

9. A. Weinberger, "Large Scale Integration of MOS Complex Logic: A Layout Method," *IEEE Journal of Solid-State Circuits*, Vol. SC-2, 1967, pp. 182-196.

10. Isao Shirakawa, Noboru Okuda, Takashi Harada, Sadahiro Tani and Hiroshi Ozaki, "A Layout System for the Random Logic Portion of an MOS LSI Chip," *IEEE Transactions on Computers*, Vol. C-30, No. 8, 1981, pp. 572-581.

11. J. Luhukay and W. J. Kublitz, "A Layout Synthesis System for nMOS Gate-cells," *19th Design Automation Conference Proceedings*, ACM/IEEE, 1982, pp. 307-314.

12. Steven P. Reiss and John E. Savage, "SLAP - A Methodology for Silicon Layout," *International Conference on Circuits and Computers Proceedings*, IEEE Computer and Circuits and Systems Societies, 1982, pp. 281-284.

13. Neil R. Quinn, Jr. and Melvin A. Breuer, "A Force Directed Component Placement Procedure for Printed Circuit Boards," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, No. 6, 1979, pp. 377-388.

14. John Newkirk, Robert Mathews, Charlie Burns and John Redford, "The Stanford nMOS Cell Library," Tech. report, Dept. of Electrical Engineering, Stanford University, 1981.